Clube Delphi

Desenvolvimento Android com Delphi
Crie um app de recolhimento de assinaturas

FireMoney e FireDAC
Como criar uma aplicação completa de Ordens de Serviço

Edição 172 :: R$ 14,90

DEVMEDIA

ISSN 1517990-7

Aplicações mestre-detalhe com FireDAC

Aprenda a utilizar os recursos de Cache Updates

Remember me

Forgot Password?

MVP

R$ 1.000.000,00
INVESTIDOS EM CONTEÚDO NOS ÚLTIMOS 12 MESES.

APLIQUE ESSE INVESTIMENTO NA SUA CARREIRA...

E MOSTRE AO MERCADO QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's consumido + de **500.000** vezes

POR APENAS
R$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

DEVMEDIA

Clube Delphi

172ª Edição 2017 ISSN 1517990-7 Impresso no Brasil

Editor Geral

Joel Rodrigues (joelrlneto@gmail.com)

Jornalista Responsável

Kaline Dolabella - JP24185

Revisora Técnica

Anny Caroline (annycarolinegnr@gmail.com)

Capa e Diagramação

Romulo Araujo

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia: **www.devmedia.com.br/mvp**

DEVMEDIA

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Joel Rodrigues - Editor da Revista
joelrlneto@gmail.com

Joel Rodrigues - **Editor Geral**

Editor da ClubeDelphi Magazine

Sumário

Artigo no estilo Solução completa

04 – Aplicações mestre-detalhe com FireDAC
[*Jones Granatyr e Fabio Spak*]

12 – Desenvolvimento Android no Delphi
[*Ubiratan Roberte*]

Artigo no estilo Curso

22 – FireMonkey e FireDAC: Construindo uma aplicação completa – Parte 1
[*Jones Granatyr*]

Aplicações mestre-detalhe com FireDAC

Aprenda a utilizar os recursos de Cache Updates do FireDAC

Quando se pensa no desenvolvimento de software seguindo a metodologia cliente e servidor, basicamente podemos utilizar dois métodos principais no que diz respeito à manipulação dos registros. O primeiro é o mais comum e ocorre quando os registros são enviados um a um para a base de dados, ou seja, a cada clique em algum botão específico é executado um comando insert, update ou delete. Por outro lado, o segundo método é utilizar conceitos de cache. Nesse caso, as atualizações são armazenadas em memória até que o usuário decida aplicá-las no servidor em uma operação em lote (batch), que geralmente ocorre em uma única transação. A segunda abordagem é bastante conhecida dos desenvolvedores Delphi, sendo que esses mecanismos são geralmente implementados utilizando dbExpress em conjunto com os componentes TClientDataSet e TDataSetProvider.

Conforme mencionado anteriormente, sem a utilização de cache os dados são enviados um por um diretamente para o banco de dados, fato que não permite a revisão dos dados por parte dos usuários ou então a gravação em lote. Portanto, a utilização desse recurso apresenta uma série de vantagens, das quais podemos citar a visualização e possibilidade de modificação dos dados antes de serem gravados e, principalmente, a facilidade de salvar os dados em um arquivo local, o que permite trabalhar com o sistema desconectado. Nesse contexto, a conexão ao servidor seria necessária somente na gravação final de todos os registros.

Apesar de possuir essas vantagens, um problema desse tipo de aplicação é que os dados podem "envelhecer" rapidamente. Imagine que dois usuários acessem um mesmo conjunto de registros e que o primeiro abra uma janela e deixe o sistema inativo por algum tempo. Enquanto isso, o segundo usuário realiza várias alterações e as grava no servidor. Neste contexto, o primeiro usuário estaria trabalhando com uma cópia inválida dos dados, o que aumentaria o risco de sobrescrever os registros e até mesmo invalidar as modificações aplicadas pelo primeiro usuário. De acordo com isso, podemos perceber que aplicações desta natureza necessitam de um maior controle para evitar inconsistência dos dados.

Baseado nisso, o objetivo do presente artigo é mostrar como construir uma aplicação mestre-detalhe utilizando o Delphi 10.1 Berlin, a engine FireDAC de acesso a dados e o SQL Server como o sistema gerenciador de banco de dados. Será mostrado como criar a base de dados e utilizar duas formas para a implementação da janela mestre-detalhe: a primeira sem a utilização dos recursos de cache e a segunda fazendo uso desse recurso. Serão também discutidas algumas vantagens de utilizar cache, sendo mostrados também vários recursos adicionais que o uso desse mecanismo possibilita. As próximas seções mostram o desenvolvimento completo do software.

Fique por Dentro

Aplicações que utilizam a relação N x N são denominadas do tipo mestre-detalhe. Nesse tipo de aplicação a exibição dos dados contidos na tabela detalhe é dependente da tabela mestre, ou seja, a seleção de dados na tabela mestre faz com que os dados exibidos na tabela detalhe sejam atualizados automaticamente. Essas aplicações são as mais comuns e largamente utilizadas em sistemas comerciais, uma vez que podem trazer agilidade e economia de tempo nas telas de consulta. Elas podem ser construídas utilizando ou não os recursos de cache, sendo que essa segunda abordagem apresenta uma série de vantagens e funcionalidades adicionais que podem aumentar o controle sobre os dados nesse tipo de aplicação, como o envio de registros em lote para serem gravados e um controle de erro mais eficaz. Desta forma, é importante o conhecimento sobre como construir aplicações mestre-detalhe utilizando a engine de acesso a dados padrão do Delphi atualmente, o FireDAC.

Criando a base de dados

O primeiro passo é a criação da base de dados que representará nosso exemplo de aplicação mestre-detalhe.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Para isso, a **Listagem 1** apresenta os comandos SQL para a definição da base de dados e das tabelas no SQL Server. Na linha 01 encontra-se o comando para a criação da base de dados, enquanto que entre as linhas 04 e 11 estão os comandos para a criação da tabela de clientes e de produtos. Como pode ser observado, essas duas tabelas armazenam somente um identificador (chave primária e autoincremento - *identity*) e um nome.

Entre as linhas 12 e 16 encontra-se o script para a definição da tabela de vendas, que, além da chave primária autoincremento, possui a data da venda e uma chave estrangeira para a tabela de clientes. Por fim, a partir da linha 17 se localiza o script de criação da tabela *venda_produtos*, que armazenará os produtos existentes em cada venda e que caracteriza a relação mestre-detalhe deste exemplo. Essa tabela contém uma chave primária autoincremento, a quantidade de produtos comprados e duas chaves estrangeiras: uma relacionada à venda e outra ao produto.

Listagem 1. Scripts de criação da base de dados.

```
01 create database mestre_detalhe;
02 go
03 use mestre_detalhe;
04 create table clientes (
05   idcliente int not null primary key identity,
06   nome varchar(50) not null
07 );
08 create table produtos (
09   idproduto int not null primary key identity,
10   nome varchar(50) not null
11 );
12 create table vendas (
13   idvenda int not null primary key identity,
14   idcliente int not null foreign key references clientes (idcliente),
15   data_venda date not null
16 );
17 create table venda_produtos (
18   idvenda_produtos int not null primary key identity,
19   idvenda int not null foreign key references vendas (idvenda),
20   idproduto int not null foreign key references produtos (idproduto),
21   quantidade int not null
22 );
```

O código da **Listagem 2** apresenta os scripts em SQL para inserção de dois clientes (linhas 01 e 02), dois produtos (linhas 03 e 04), duas vendas (linhas 05 e 06) e três produtos nas vendas (linhas 07 a 09). A primeira venda possui somente um produto cadastrado, enquanto que a segunda contém dois. É importante salientar que, pelo fato dos campos chave primária serem autoincremento, não é necessário passar seus valores na cláusula *values* dos comandos *insert*.

Definindo o acesso aos dados

O próximo passo é criar uma nova aplicação do tipo VCL no Delphi (*File > New > VCL Forms Application – Delphi*), salvar o formulário gerado pela IDE como *PrincipalFrm* e nomeá-lo (pela propriedade *name*) como *frmPrincipal*. Esse tipo de aplicação utiliza a biblioteca *Visual Component Library* (VCL) e diz respeito a softwares que funcionarão somente no Windows, diferentemente de aplicações criadas com o FireMonkey, que são multiplataforma.

Listagem 2. Scripts para inserção de dados.

```
01 insert into clientes (nome) values ('Ana');
02 insert into clientes (nome) values ('Pedro');
03 insert into produtos (nome) values ('Suco');
04 insert into produtos (nome) values ('Pão');
05 insert into vendas (idcliente, data_venda) values (1, '2017-01-01');
06 insert into vendas (idcliente, data_venda) values (2, '2017-01-02');
07 insert into venda_produtos (idvenda, idproduto, quantidade) values (1, 1, 5);
08 insert into venda_produtos (idvenda, idproduto, quantidade) values (2, 1, 3);
09 insert into venda_produtos (idvenda, idproduto, quantidade) values (2, 2, 2);
```

Todos os componentes de acesso aos dados serão centralizados em um Data Module. Para criá-lo deve-se escolher a opção de menu *File > New > Other > Delphi Files > Data Module*, salvá-lo com o nome *ConexaoDtm* e nomeá-lo, através de suas propriedades, como *dtmConexao*. A **Figura 1** apresenta os componentes que serão utilizados, sendo: um TFDConnection (*cnnConexao*), dois TFDQuery (*qryVendas* e *qryVendaProdutos*), um TFDMemTable (*memCache*), três TDataSource (*dsVendas*, *dsVendaProdutos* e *dsCache*) e um TFDSchemaAdapter (*adapter*).

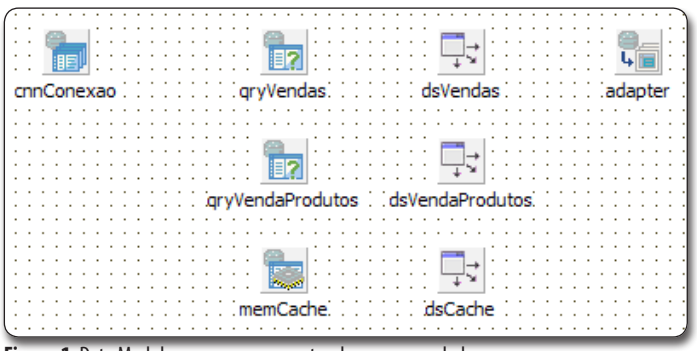

Figura 1. Data Module com os componentes de acesso aos dados

A primeira configuração que faremos é a conexão do sistema com a base de dados por meio do componente *cnnConexao*. Basta dar um duplo clique nesse componente para que a janela *FireDAC Connection Editor* seja aberta, na qual as seguintes propriedades devem ser configuradas:

- Driver ID: *MSSQL*
- Database: *mestre_detalhe*
- Server: *local*
- OSAuthent: *Yes*

As três últimas propriedades são relativas ao nome da base de dados, à localização do servidor e à forma de autenticação, respectivamente. Note que não foi necessário configurar o nome do usuário e senha de acesso à base de dados visto que neste exemplo estamos utilizando a segurança integrada do SQL Server, ou seja, são usadas as mesmas credenciais de login do sistema operacional.

A *qryVendas* e a *qryVendaProdutos* têm a função de extrair os dados da base de dados e efetivamente vincular as duas tabelas para que o relacionamento mestre-detalhe seja possível (ambas devem estar ligadas ao *cnnConexao* por meio da propriedade *Connection*).

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

A primeira query deve ter o seguinte comando na propriedade *SQL*, o qual somente extrai todos os registros da tabela mestre (*vendas*):

```
select idvenda, idcliente, data_venda from vendas
```

Por outro lado, a segunda query deve ter o script a seguir configurado em sua propriedade *SQL*:

```
select idvenda_produtos, idvenda, idproduto, quantidade from venda_produtos where
idvenda = :idvenda
```

Esse comando é bastante similar ao anterior e é responsável por trazer todos os registros da tabela detalhe (*venda_produtos*). A principal diferença está no fato de existir um comando *where* no final do script, que de fato realiza o relacionamento entre as tabelas mestre e detalhe. Note que para essa cláusula é utilizado o campo *idvenda* da tabela *venda_produtos*, o qual representa a chave estrangeira que faz a ligação com a tabela mestre. Outro fato importante é a existência de um parâmetro (*:idvenda*), que deve possuir exatamente o mesmo nome do campo chave primária da tabela mestre, caso contrário o relacionamento automático entre as tabelas pode não ocorrer de maneira adequada. Não há necessidade de preenchimento manual via código desse parâmetro, pois o Delphi identificará automaticamente o relacionamento e preencherá o parâmetro em tempo de execução de acordo com os registros que forem selecionados pelo usuário na interface gráfica do sistema (a ser construída na sequência).

Com relação aos dois componentes TDataSource (*dsVendas* e *dsVendaProdutos*), ambos devem ter suas propriedades *DataSet* setadas para *qryVendas* e *qryVendaProdutos*, respectivamente. Essa ligação tem o objetivo de possibilitar acesso à ligação mestre-detalhe, bem como permitir que os dados sejam exibidos em tela posteriormente. Além da associação entre os componentes por meio de SQL, é necessário ainda configurar as seguintes propriedade para a *qryVendaProdutos*:

- MasterSource: *dsVendas*
- MasterFields: *idvenda*

Essas duas propriedades estão efetivamente ligadas ao relacionamento entre os componentes e é por meio delas que o Delphi entenderá o parâmetro *idvenda* e de fato operacionalizará a ligação entre as tabelas. Adicionalmente, deve-se clicar com o botão direito nas queries, acessar a opção *Fields Editor* e escolher *Add All Fields* para que todos os campos sejam adicionados.

Com relação aos outros componentes presentes na **Figura 1**, a única configuração que precisa ser feita é para o DataSource *dsCache*, que deve ter sua propriedade *DataSet* alterada para *memCache*. Mais detalhes sobre esses componentes serão apresentados na sequência.

Criando a janela mestre-detalhe sem cache

A **Figura 2** apresenta a interface gráfica que usaremos para realizar os testes em nossa aplicação, a qual possui uma TToolBar (*tblFuncoes*) com sete TButtons (*btnVerAlteracoes, btnUltimaAlteracao, btnReverter, btnCancelar, btnCriarSavePoint, btnRestaurarSavePoint* e *btnGravar*). Logo abaixo, existem três TDBGrids (*grdVendas, grdVendaProdutos* e *grdCache*), os quais exibirão respectivamente: as vendas, os produtos somente das vendas selecionadas na primeira grade e os dados que estão armazenados no cache. Os dois primeiros dizem respeito ao relacionamento mestre-detalhe, enquanto que a terceira grade será utilizada para entendermos as funcionalidades dos processos de cache no Delphi. Essas funcionalidades serão explanadas por meio das funções apresentadas nos botões.

É necessário, agora, vincular os DBGrids aos seus respectivos DataSources presentes no Data Module. Para isso, deve-se primeiramente seguir a opção *File > Use Unit* e escolher o arquivo *ConexaoDtm.pas* para que a respectiva unit seja adicionada na cláusula *uses* do formulário, possibilitando acesso aos componentes. Por fim, cada grade necessita ter sua propriedade *DataSource* configurada de modo que esteja ligada às tabelas correspondentes.

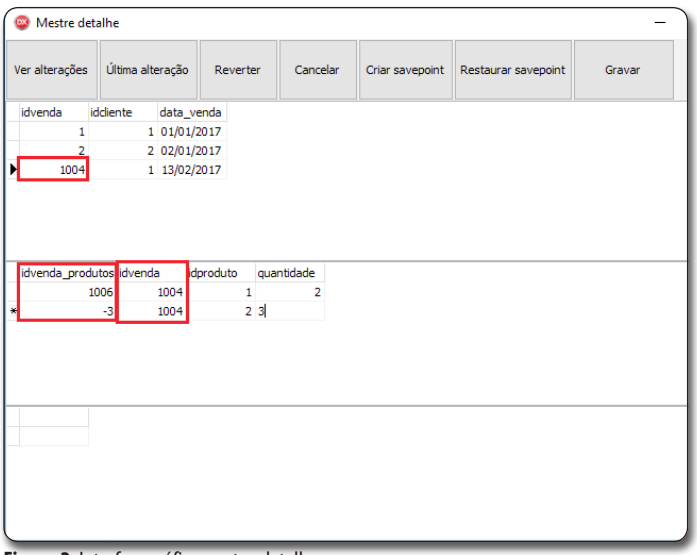

Figura 2. Interface gráfica mestre-detalhe

A partir desse momento a aplicação já pode ser executada, bastando que a propriedade *Active* das duas queries esteja configurada com o valor *True*. Neste exemplo não estamos utilizados botões para inclusão, alteração e exclusão de registros. Dessa forma, para inclusão deve-se somente pressionar a seta para baixo do teclado e para exclusão, o conjunto das teclas CTRL + DELETE.

A **Figura 2** apresenta a simulação da inserção de uma nova venda juntamente com dois produtos, podendo-se observar que, após gravada, a nova venda recebeu o identificador 1004, que corresponde ao campo autoincremento/chave primária do tipo *identity* utilizado pelo SQL Server. É importante salientar que neste exemplo o valor 1004 refere-se ao próximo identificador disponível nesta base de dados, podendo variar de base para base de acordo com a quantidade de registros. Esse mecanismo de autoincremento é gerenciado pelo próprio banco de dados e o programador não tem muito controle sobre os valores gerados.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Note também que no segundo registro da segunda grade está presente o valor -3, que indica a chave primária temporária até que o valor seja gravado em definitivo. O FireDAC utiliza essa estratégia de valores decrementais negativos para evitar conflito com possíveis restrições de integridade; por exemplo, se forem inseridos cinco registros, os valores estarão no intervalo de -1 até -5. É também possível observar na segunda grade que a chave estrangeira *idvenda* foi preenchida automaticamente com o valor da chave primária da tabela mestre para efetivamente concretizar o relacionamento entre ambas.

Nota

A própria engine do FireDAC detecta automaticamente os campos que foram definidos como autoincremento na base de dados, independente do sistema gerenciador de banco de dados utilizado, desde que esse seja suportado (consulte a seção Links para mais informações sobre os outros SGBDs suportados). Abrindo o Fields Editor de qualquer uma das queries utilizadas (botão direito no componente) e clicando no campo chave primária, pode-se notar que a propriedade ServerAutoIncrement já vem marcada como True, o que indica que o próprio servidor é responsável por gerenciar o incremento dos campos e que o FireDAC faz a interface entre o SGBD e a aplicação. Esse tipo de campo é tratado de maneira diferente de acordo com o banco de dados, por exemplo: no Firebird e Interbase são utilizados Generators e Triggers, no Informix e PostgreSQL são os campos do tipo serial, no MySQL é usado o tipo auto_increment e no Sybase e SQL Server é o tipo identity. A vantagem deste recurso do FireDAC é que a engine consegue manipular esses tipos de campos por mais que eles apresentem maneiras diferentes de serem construídos.

Nota

No que diz respeito a utilização de campos do tipo autoincremento, esses podem ser definidos diretamente no banco de dados ou então na própria aplicação. A abordagem recomendada é a primeira, pois ela é de responsabilidade direta do SGBD e está fora das transações da aplicação. Em algumas situações, não é uma boa prática manter um contador na aplicação porque ele está em uma transação e os outros usuários não terão acesso até que ela seja "comitada". Essa abordagem garante o controle de colisão, evitando que vários usuários recebam códigos repetidos.

Com isso, sem nenhuma linha de código é possível construir uma interface totalmente funcional. Porém, antes de cadastrar um novo produto a venda precisa ser gravada, ou seja, a chave primária da venda precisa ser gerada para que ela seja copiada para a tabela detalhe (utilizados os recursos de *post* do TFDQuery). Isso pode ser observado novamente na **Figura 2**, na qual é possível notar que o valor 1004 foi gerado na primeira grade para depois ser copiado para a segunda.

Esse tipo de comportamento pode ser útil e adequado em várias situações, todavia, em alguns casos pode gerar alguns problemas. Para exemplificar, consideremos o exemplo de um sistema on-line de nota fiscal, no qual o cabeçalho da nota (tabela mestre) necessita ser enviado para o servidor junto com todos os produtos que foram comprados (tabela detalhe). Da forma que o sistema está implementado, até agora isso não seria possível, pois primeiramente o cabeçalho seria enviado para o servidor e depois cada um dos produtos individualmente. Caso ocorresse algum problema de conexão, a nota fiscal poderia ficar gravada no servidor somente com o cabeçalho ou então com menos produtos do que efetivamente teria.

Criando a janela mestre-detalhe com cache

Para resolver a questão elencada anteriormente, serão apresentados na sequência os recursos de cache updates, os quais permitirão que os dados sejam salvos localmente por padrão e enviados em lote para o servidor somente após um comando realizado pelo usuário. Esse recurso é muito semelhante ao utilizado em aplicações dbExpress, as quais utilizam TClientDataSet e TDataSetProvider, porém aqui serão utilizados os recursos da engine FireDAC para acesso a dados.

Para podermos trabalhar com dados em cache, na query mestre (*qryVendas*) a única propriedade que precisa ser configurada é a *CachedUpdates*, que passa a conter o valor *True*. De forma similar, na query detalhe (*qryVendaProdutos*) essa mesma propriedade deve ser mudada, além de setar para *True* a propriedade *FetchOptions* > *DetailCascade* e preencher *IndexFieldNames* com o valor *idvenda*, que equivale ao campo de chave estrangeira. A propriedade *DetailCascade*, por sua vez, controla a propagação de um conjunto de dados mestre para um conjunto de dados de detalhe, devendo ser utilizada quando está sendo aplicado o modo de atualização de cache centralizado, que será visto adiante com o componente TFDSchemaAdapter.

A **Figura 3** apresenta a inserção de uma venda e dois produtos, na qual é possível observar que o campo autoincremento utilizado tanto no mestre quanto no detalhe é uma chave primária temporária com o valor "-1", o que indica que os dados ainda não foram gravados. Para efeitos de comparação, observe novamente que na **Figura 2** os registros detalhes receberam o valor "1004", que diz respeito ao campo autoincremento gravado antes dos detalhes serem inseridos.

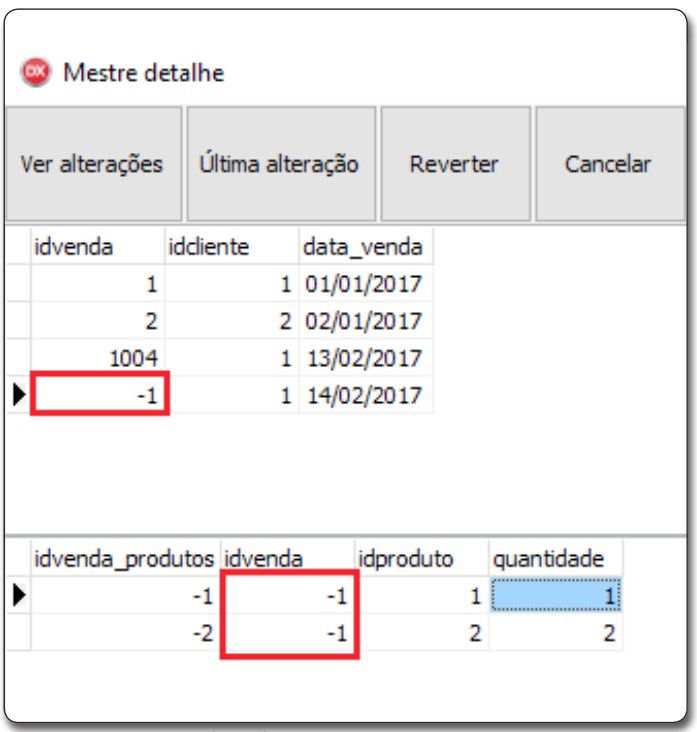

Figura 3. Inserção com cache updates

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Desta forma, é necessário fazer uso do comando *ApplyUpdates* para que os três novos registros da **Figura 3** sejam gravados em conjunto na base de dados em um único lote. Os componentes do tipo TFDQuery disponibilizam esse método, porém utilizaremos o TFDSchemaAdapter (*adapter*) (**BOX 1**) localizado no Data Module pelo fato de ele facilitar o gerenciamento da gravação. Para isso, as duas queries (*qryVendas* e *qryVendaProdutos*) devem ter sua propriedade *SchemaAdapter* configuradas para o componente correspondente: *adapter*.

BOX 1. TFDSchemaAdapter

O componente TFDSchemaAdapter é utilizado para gerenciar a gravação centralizada quando o recurso de cache updates é utilizado. Esse recurso oferece a facilidade de utilizar somente um comando ApplyUpdates quando vários DataSets estão presentes na aplicação, servindo como um armazenamento central para os registros e suas alterações em vários conjuntos de dados.

Depois dos registros terem sido gravados é necessário fazer uma limpeza no cache — o código apresentado na **Listagem 3** apresenta a definição de um procedimento que realiza essa tarefa. Na linha 02 foi definida a assinatura do método na seção *private* da unit *ConexaoDtm*, sendo que para ter sua implementação gerada deve-se pressionar o conjunto de teclas CTRL + SHIFT + C. Nas linhas 06 e 07 é executado o método *CommitUpdates* para as duas queries, que tem a função de limpar o log de alterações e marcar todos os registros com o status "não modificado". Esse método é comumente invocado depois de uma chamada com sucesso ao *ApplyUpdates*.

A **Listagem 4** apresenta a codificação que deve ser feita no evento *OnCreate* do Data Module. Na linha 03 o evento *AfterApplyUpdate* do SchemaAdapter recebe o método *LimpaCache*, o que garante que toda vez que uma gravação acontecer o cache será limpo logo após (caso essa limpeza do cache não seja efetuada, a cada clique no botão gravar os registros serão duplicados). Adicionalmente, nas linhas 04 e 05 as duas queries são abertas para que não seja necessário manter a propriedade *Active* = *True* em modo de design.

Listagem 3. Método para limpar o cache das queries.

```
01 private
02   procedure LimpaCache(Sender: TObject);
03 public
04 procedure TdtmConexao.LimpaCache(Sender: TObject);
05 begin
06   qryVendas.CommitUpdates;
07   qryVendaProdutos.CommitUpdates;
08 end;
```

Listagem 4. Evento OnCreate do Data Module.

```
01 procedure TdtmConexao.DataModuleCreate(Sender: TObject);
02 begin
03   adapter.AfterApplyUpdate := LimpaCache;
04   qryVendas.Open();
05   qryVendaProdutos.Open();
06 end;
```

Por fim, a **Listagem 5** apresenta o código que deve ser digitado no evento *OnClick* do botão gravar (*btnGravar*) do formulário principal. Na linha 03 é invocado o método *ApplyUpdates* do *adapter*, gravando tanto os registros da tabela mestre quanto da tabela detalhe utilizando uma única linha de código. Conforme mencionado anteriormente, o uso do componente SchemaAdapter centraliza a gravação de registros de várias tabelas, fato que facilita o controle de erros e principalmente evita que alguns detalhes sejam inseridos e outros não.

Ainda na linha 03, o valor "0" indica que nenhum erro pode ocorrer em toda a gravação, ou seja, caso estejamos incluindo uma venda com dez produtos e ocorra erro somente em um deles, toda a operação será cancelada. Esse recurso garante a integridade dos dados se levarmos em consideração o exemplo da gravação de uma nota fiscal completa explanado anteriormente.

Listagem 5. Gravação dos dados com SchemaAdapter e ApplyUpdates.

```
01 procedure TfrmPrincipal.btnGravarClick(Sender: TObject);
02 begin
03   dtmConexao.adapter.ApplyUpdates(0);
04 end;
```

Para uma melhor visualização e entendimento de como o cache no FireDAC funciona, vamos agora fazer uso do componente TFDMemTable. Esse componente será responsável por armazenar e exibir na terceira grade do formulário (*grdCache*) todas as modificações feitas em memória na tabela mestre antes de serem aplicadas na base de dados.

Para isso, a **Listagem 6** apresenta o código que deve ser implementado no evento *FormActivate* do formulário principal. Na linha 03 é executado o método *CloneCursor* do memory table, passando como parâmetro a query mestre (*qryVendas*). Em resumo, esse código cria um clone do cursor da *qryVendas* para o *memCache*, de modo que o novo cursor aponte para os mesmos registros do cursor da *qryVendas*.

Listagem 6. Evento FormActivate do formulário principal.

```
01 procedure TfrmPrincipal.FormActivate(Sender: TObject);
02 begin
03   dtmConexao.memCache.CloneCursor(dtmConexao.qryVendas, true);
04   dtmConexao.memCache.FilterChanges := [rtModified, rtInserted, rtDeleted];
05 end;
```

Quando esse método é invocado todos os registros serão exibidos na tabela em memória, porém o que nos interessa neste momento é a visualização somente de alterações, inserções e exclusões no cache. Desta forma, na linha 04 esse filtro é feito por meio da propriedade *FilterChanges*. O comando *rtModified* é responsável pelos registros alterados após a busca ou após a última operação de um *CommitUpdates* ou *CancelUpdates*. Já o *rtInserted* mostra os novos registros, ainda não lançados no banco de dados e

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

adicionados depois que o conjunto de dados foi aberto ou após a última operação *CommitUpdates*. O comando *rtDeleted* mostra os registros que foram excluídos na aplicação, mas não do banco de dados. Ainda é possível utilizar o *rtUnmodified*, que, por sua vez, mostra os registros inalterados, e também o *rtHasErrors*, cuja função é verificar os registros que possuem erros associados após as operações de *ApplyUpdates*.

O objetivo é que a tabela em memória exiba um status sobre que tipo de operação ocorreu em cada registro, ou seja, se foi uma alteração, inserção ou exclusão. Essa informação ficará armazenada em um campo calculado. Para criá-lo, deve-se acessar o *Fields Editor* da tabela em memória (*memCache*) clicando com o botão direito no componente e depois escolher a opção *New Field* (botão direito também). A **Figura 4** apresenta a janela correspondente, na qual foi definido o campo *status*, do tipo *String*, com tamanho 20 e do tipo calculado (*Calculated*). Note que também estão presentes os campos *idvenda*, *idcliente* e *data_venda*, que devem ser copiados do *Fields Editor* da *qryVendas* e colados no *Fields Editor* da tabela em memória.

Nota

O componente TFDMemTable é um DataSet em memória e é utilizado para gerenciar registros no lado cliente da aplicação e também para trocar dados com uma base de dados. Tem funcionalidades semelhantes ao TClientDataSet, mas segue o modelo Cache Updates. Ou seja, quando o valor da propriedade CachedUpdates for True, o TFDMemTable irá gravar as alterações no log de atualizações. Para que essas atualizações sejam gravadas no banco de dados é necessário utilizar o comando ApplyUpdates e, em seguida, para mesclar o log de alterações, chamar o comando CommitUpdates. Esse componente é comumente utilizado em aplicações móveis que empregam a tecnologia DataSnap.

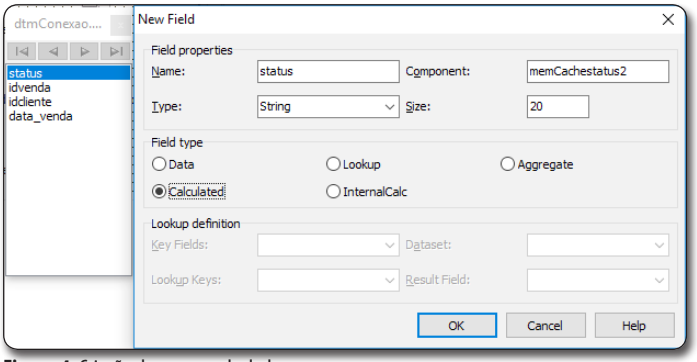

Figura 4. Criação de campo calculado

Um último passo para que possamos visualizar as alterações de cache é programar o evento *CalcFields* da tabela em memória (*memCache*) conforme o mostrado na **Listagem 7**. Na linha 03 é verificado se o status de atualização do DataSet é de alteração (*usModified*). Caso positivo o campo status recém-criado recebe o valor *Modificado*, que será exibido no formulário. O mesmo processo segue no restante do código, ou seja, na linha 05 é feita a verificação de inserções e na linha 07 de exclusões.

A **Figura 5** apresenta uma simulação do armazenamento do cache na tabela em memória, podendo ser visto na parte inferior da figura. O primeiro registro (*idvenda = 1*) teve a data alterada e, por isso, foi mostrado com o valor *Modificado* no status, enquanto que o segundo (*idvenda = 2*) foi apagado e seu status mostra o valor *Excluído* (a venda recém-excluída é mostrada somente na grade inferior). Por fim, um novo registro foi inserido na grade superior (*idvenda = -1*) e é exibido com o status *Inserido*. Com essa funcionalidade, é possível observar e controlar em tempo real todas as operações que são executadas em memória para que, assim, tratamentos ou correções específicas possam ser realizadas quando necessário antes dos dados serem efetivamente gravados.

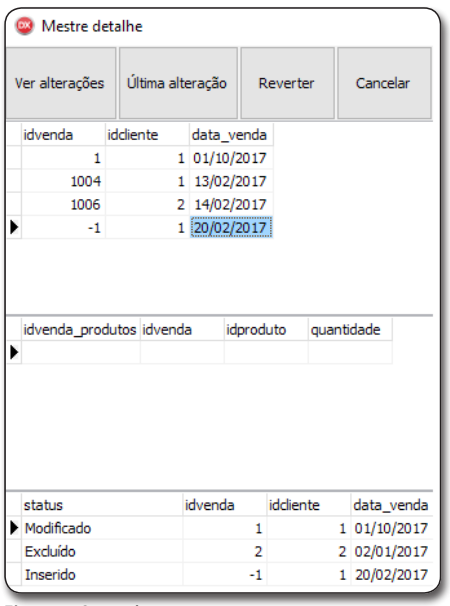

Figura 5. Status dos registros

Listagem 7. Evento CalcFields para exibir o status.

```
01 procedure TdtmConexao.memCacheCalcFields(DataSet: TDataSet);
02 begin
03   if DataSet.UpdateStatus = usModified then
04     DataSet.FieldByName('status').AsString := 'Modificado'
05   else if DataSet.UpdateStatus = usInserted then
06     DataSet.FieldByName('status').AsString := 'Inserido'
07   else if DataSet.UpdateStatus = usDeleted then
08     DataSet.FieldByName('status').AsString := 'Excluído';
09 end;
```

Para apresentar um controle mais detalhado dos erros que podem ocorrer no processo de gravação, a **Listagem 8** apresenta o código que deve ser adicionado no evento *OnClick* do botão *btnGravar* (em substituição da codificação apresentada na **Listagem 5**). Na linha 03 é declarada a variável *AErros*, que tem a função de armazenar a quantidade de erros que ocorreram no processo de gravação dos dados (linha 05). Note que foi utilizado o valor "-1" na linha 05, que indica que o FireDAC deve tentar gravar todos os registros que conseguir, independentemente de serem gerados erros, enquanto que o uso do valor "0" na **Listagem 5** quer dizer que nenhum erro é permitido.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Vale citar também que se for passado como parâmetro um inteiro positivo, o número de erros permitido será igual a esse parâmetro. Na linha 06 é verificado se a quantidade de erros é maior do que zero, e caso positivo o filtro da tabela em memória é alterado para o valor *rtHasErrors* (linha 07), ou seja, serão mostrados na última grade os registros que foram recusados pelo servidor e que devem ser revisados pelo usuário. Por fim, na linha 08 o cache da query de vendas é limpo.

Listagem 8. Gravação dos dados com tratamento de erros.

```
01 procedure TfrmPrincipal.btnGravarClick(Sender: TObject);
02 var
03   AErros: Integer;
04 begin
05   AErros := dtmConexao.adapter.ApplyUpdates(-1);
06   if AErros > 0 then
07     dtmConexao.memCache.FilterChanges := [rtHasErrors];
08   dtmConexao.qryVendas.CommitUpdates;
09 end;
```

Outras funcionalidades

Nesta seção veremos outros recursos bastante úteis quando trabalhamos com cache, que são: visualização dos valores modificados, reversão de alterações, cancelamento de todas as modificações e uso de *save points*. Cada uma dessas funções corresponde a um botão do formulário.

A primeira função que será abordada é a visualização dos novos e antigos valores de determinados campos. A **Listagem 9** apresenta o código que deve ser implementado no evento *OnClick* do botão *btnVerAlteracoes*. Conforme pode ser observado na linha 03, o valor antigo é dado pela propriedade *OldValue*, enquanto que o valor que o usuário modificou é dado pela propriedade *Value*.

É importante frisar que esse código está mostrando alterações somente no campo *idcliente*. Para testá-lo, basta trocar o valor desse campo, manter o registro selecionado e clicar no botão correspondente. Esse recurso pode ser interessante em aplicações nas quais é necessário realizar validações em registros que estão sendo alterados, por exemplo em um controle de estoque. Nesse cenário, uma validação pode ser executada para verificar se a nova quantidade informada pelo usuário está de acordo com a quantidade de produtos existentes no estoque, assim evitando que seja vendida uma quantidade maior do que a efetivamente disponível. De forma similar, caso o usuário diminua a quantidade, pode-se incrementar os valores no estoque.

Entre as linhas 01 e 04 da **Listagem 10** encontra-se o código que deve ser implementado no evento *OnClick* do botão *btnUltimaAlteracao*, enquanto que entre as linhas 05 e 08 está o código que corresponde ao evento *OnClick* do botão *btnReverter*. O método *UndoLastChange* da linha 03, como o próprio nome sugere, tem o objetivo de reverter as últimas alterações realizadas. Supondo-se que três registros sejam alterados, se o botão for clicado três vezes cada um dos cliques reverterá os dados somente de um único registro, muito semelhante ao comando desfazer (CTRL + Z) comumente utilizado em processadores de texto, por exemplo. Já o método *RevertRecord* (linha 07) desfaz as alterações somente no último registro modificado, ou seja, se três registros forem alterados somente o último voltará aos valores originais. Podemos resumir a diferença entre os dois comandos da seguinte forma: o primeiro permite a restauração de vários registros (um por clique), enquanto que o segundo possibilita a restauração somente da última modificação.

Outro comando utilizado para cancelar alterações é o *CancelUpdates*, que restaura as modificações em todos os registros que ainda não foram gravados, diferentemente dos dois métodos anteriores, que fazem a restauração somente de um registro por vez. A **Listagem 11** apresenta o código correspondente (linha 03), que deve ser implementado no evento *OnClick* do botão *btnCancelar*. Essa opção é particularmente útil quando várias alterações estão sendo feitas nos itens de uma nota fiscal, por exemplo, e, por alguma razão, todas as alterações devem ser canceladas ao mesmo tempo.

Listagem 9. Visualização dos valores modificados.

```
01 procedure TfrmPrincipal.btnVerAlteracoesClick(Sender: TObject);
02 begin
03   ShowMessage('Valor antigo: ' + IntToStr(dtmConexao.qryVendas.
FieldByName('idcliente').OldValue) +
   ' Valor novo: ' + IntToStr(dtmConexao.qryVendas.FieldByName('idcliente').Value));
04 end;
```

Listagem 10. Comandos UndoLastChange e RevertRecord.

```
01 procedure TfrmPrincipal.btnUltimaAlteracaoClick(Sender: TObject);
02 begin
03   dtmConexao.qryVendas.UndoLastChange(True);
04 end;
05 procedure TfrmPrincipal.btnReverterClick(Sender: TObject);
06 begin
07   dtmConexao.qryVendas.RevertRecord;
08 end;
```

Listagem 11. Cancelamento de todas as alterações.

```
01 procedure TfrmPrincipal.btnCancelarClick(Sender: TObject);
02 begin
03   dtmConexao.qryVendas.CancelUpdates;
04 end;
```

O último exemplo a ser abordado sobre as funcionalidades que podem ser implementadas com cache updates é a utilização de save points, que são pontos de "marcação" que o usuário pode escolher durante a manipulação dos registros, podendo restaurar todos os dados a partir do ponto marcado. Por exemplo, se um usuário cria um save point e faz a modificação de cinco registros, no momento em que pedir para restaurar o save point o FireDAC restaurará exatamente esses cinco registros que foram "marcados" no início.

Esse recurso é útil em situações nas quais o usuário necessita manipular registros, mas não há certeza sobre os novos dados informados. Desta forma, o usuário poderá adicionar um save

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

point quando iniciar as alterações e, caso algum erro aconteça, basta restaurar a partir do ponto marcado. A **Listagem 12** apresenta a codificação necessária para implementar esse recurso, sendo que na linha 02 é declarada a variável global *FSavePoint*, na seção *private* do formulário, a qual será utilizada para realizar o controle do ponto inicial.

Na linha 05 encontra-se o código para a criação do save point, que deve ser programado no evento *OnClick* do botão *btnCriarSavePoint* (linha 03). Similarmente, na linha 09 está o código responsável pela restauração dos dados, que deve ser implementado no evento *OnClick* do botão *btnRestaurarSavePoint*. Conforme pode ser observado na listagem, o código é bastante simples e são utilizadas somente duas linhas: na linha 05, a variável *FSavePoint* recebe a propriedade *SavePoint* da query para realizar a marcação. Por outro lado, na linha 09 ocorre o inverso, ou seja, a propriedade *SavePoint* recebe o valor da variável inteira, o que indica que os dados devem ser restaurados para o momento em que o save point foi criado.

Listagem 12. Criação e restauração de save point.

```
01 private
02   FSavePoint: Integer;
03 procedure TfrmPrincipal.btnCriarSavePointClick(Sender: TObject);
04 begin
05   FSavePoint := dtmConexao.qryVendas.SavePoint
06 end;
07 procedure TfrmPrincipal.btnRestaurarSavePointClick(Sender: TObject);
08 begin
09   dtmConexao.qryVendas.SavePoint := FSavePoint;
10 end;
```

Com isso, este aplicativo mestre-detalhe já está funcional, e com vários recursos importantes e comumente utilizados em aplicações comerciais implementados. Conforme pudemos observar, a utilização dos recursos de cache em aplicações mestre-detalhe agrega uma série de recursos adicionais e formas diferenciadas de controle dos dados.

Muitos desenvolvedores, principalmente os que estão habituados com a engine dbExpress de acesso a dados, usualmente têm dúvida de como utilizar o FireDAC com recursos de cache semelhantes ao que já é bem conhecido da comunidade Delphi com a utilização de TClientDataSet em conjunto com o TDataSetProvider. Conforme visto neste artigo, o próprio componente TFDQuery já possui todos os recursos necessários para a construção deste tipo de sistema.

Além do conteúdo abordado nos tópicos anteriores, algumas melhorias neste aplicativo são necessárias, como o controle mais apurado de erros e, principalmente, a gerência da concorrência para evitar que dados desatualizados sejam sobrescritos. Esse último recurso pode ser implementado por meio de um status nos registros, de modo que esse campo possa ser utilizado para gerenciar se o registro pode ou não ser aberto caso algum usuário já esteja realizando modificações.

Outra dúvida bastante comum é como trabalhar com atualizações utilizando comandos "SQL complexos", que são aqueles que apresentam muitos joins com outras tabelas. Para isso, uma alternativa é a utilização do componente TFDUpdateSQL, no qual podem ser definidas instruções SQL personalizadas. Por fim, um último comentário: caso sejam necessárias mais tabelas de detalhe na aplicação, o mesmo procedimento deve ser seguido, ou seja, com o FireDAC é possível trabalhar com múltiplas tabelas mestre-detalhe.

Autor

Jones Granatyr
Doutorando em Informática bolsista CAPES e Mestre em Ciência da Computação bolsista CNPq, ambos na área de Inteligência Artificial. É fundador do portal IA Expert e trabalha em projetos de pesquisa relacionados à área de Inteligência Artificial, tais como Sistemas Especialistas, Mineração de Dados, Mineração de Textos e Sistemas Multiagente.

Autor

Fabio Spak
Pós-graduado em Business Intelligence pela Universidade Positivo. Bacharel em Ciência da Computação pela Universidade do Contestado - UnC, no campus de Porto União/SC. Desenvolve trabalhos e artigos técnicos/científicos na área de Inteligência Artificial, especificamente na linha de pesquisa de Mineração de Dados e Sistemas de Recomendação.

Links:

Documentação da Embarcadero
http://docwiki.embarcadero.com

Caching Updates (FireDAC)
http://docwiki.embarcadero.com/RADStudio/Berlin/en/Caching_Updates_(FireDAC)

TFDMemTable (FireDAC)
http://docwiki.embarcadero.com/Libraries/Berlin/en/FireDAC.Comp.Client.TFDMemTable

Filter Changes (FireDAC)
http://docwiki.embarcadero.com/Libraries/Berlin/en/FireDAC.Comp.Client.TFDMemTable.FilterChanges

FireDAC Cached Updates e Campos Auto-Inc
https://www.youtube.com/watch?v=6-W5Q_hd6j0&t=1313s

Auto-Incremental Fields (FireDAC)
http://docwiki.embarcadero.com/RADStudio/Berlin/en/Auto-Incremental_Fields_(FireDAC)

Handling Errors (FireDAC)
http://docwiki.embarcadero.com/RADStudio/Berlin/en/Handling_Errors_(FireDAC)

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Desenvolvimento Android no Delphi

Desenvolva um sistema de recolhimento de assinaturas utilizando o RAD Studio 10 Seattle

Segundo dados do Comitê Gestor de Internet no Brasil (CGI-Brasil), aproximadamente 81,5 milhões de pessoas acessam a internet através do celular, portanto faz sentido dar atenção ao desenvolvimento de aplicações mobile. Porém, os diversos sistemas operacionais mobile presentes no mercado (iOS, Android, etc.) complicam o desenvolvimento, já que, se o objetivo for atingir todos os SOs (ou a maioria), se torna necessário construir uma aplicação para cada sistema alvo, cada uma desenvolvida com uma linguagem de programação distinta.

Dessa forma, surgem ferramentas que permitem a criação de programas para diversas plataformas usando um só código e diferentes compiladores nativos. Com isso não precisamos mais ser especialistas em uma ferramenta nem ter conhecimentos em HTML 5 ou JavaScript.

A ferramenta escolhida para este artigo, o RAD Studio 10 Seattle — da Embarcadero —, permite o desenvolvimento para as três principais plataformas móveis utilizadas no país (Google Android, iOS e Windows). Porém, como cerca de 91% das pessoas utilizam aparelhos com alguma versão do Android, este artigo focará no desenvolvimento para o sistema operacional do Google.

Fique por Dentro

Muitas empresas necessitam de profissionais capacitados a desenvolver aplicações móveis, sendo que muitas delas ainda utilizam o Delphi como principal ferramenta de desenvolvimento, principalmente devido a sua facilidade de uso e possibilidade de compilação para diversas plataformas distintas sem grandes necessidades de alteração no código já escrito. Dessa forma, este artigo tem como objetivo demonstrar a criação de uma aplicação mobile completa, a qual permite o recebimento de assinaturas dos usuários e o envio das mesmas por e-mail.

A aplicação de recolhimento de assinaturas

É muito comum que empresas prestadoras de serviços dos mais variados setores requeiram de seus usuários a assinatura de documentos que comprovem a finalização e concordância do usuário com o serviço realizado (muito comum com entregas expressas). Em geral, essas assinaturas são recolhidas em documentos pré-impressos, sendo que muitas das vezes é preciso que uma via desse documento seja entregue ao usuário do serviço e outra permaneça em posse da prestadora.

No entanto, papéis, além de não serem a melhor opção para armazenamento de uma informação, por permitirem fácil extravio e serem muito suscetíveis a danos, geram volume físico. Com o passar do tempo, esse crescente volume se torna um problema, requerendo cada vez mais espaço, exigindo maiores investimentos em meios de depósito e gerando, portanto, gastos. Logo, substituir esses documentos pelos seus correspondentes em meio digital torna-se uma solução viável no curto, médio e longo prazo.

Ainda nesse contexto, ressaltamos que exigir do usuário que o mesmo possua certificado digital para assinatura do documento não se mostra uma solução viável, já que pode custar caro principalmente para quem está começando, devido à burocracia. A proposta mais aceitável passa a ser, então, o desenvolvimento de um sistema que permita ao usuário realizar a assinatura a partir da tela do dispositivo, como se estivesse assinando em papel. Essa é uma técnica já utilizada por algumas empresas e que, a partir da leitura deste artigo, poderá ser oferecida também pelo leitor.

Configurando o emulador

É possível compilar e executar o código desenvolvido em um emulador ou diretamente em um dispositivo, desde que esse esteja conectado ao computador. Para iniciar a configuração do emulador, utilizando o sistema operacional Windows, clique em *Iniciar > Todos os Programas > Embarcadero RAD Studio 10 Seattle > Android AVD Manager*.

Na tela que será aberta selecione a aba **Device Definitions**, onde serão apresentados vários emuladores que são instalados junta-

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

mente com o RAD Studio. Selecione o **Galaxy Nexus** (ou qualquer outro emulador de sua preferência), como pode ser observado na **Figura 1**, e clique em **Create AVD**.

Depois de realizada a instalação, será apresentada uma tela para configuração do emulador. Mantenha a maior parte das informações inalteradas — as configurações básicas de qualquer emulador criado já nos permitem realizar as atividades que precisamos sem problemas. Entretanto, algumas devem ser configuradas, tais como: *skin* (procurando o layout que melhor se enquadre na resolução de sua tela), *internal storage* (memória interna, permitindo que suas aplicações sejam armazenadas sem problemas) e *SD Card* (define o espaço do cartão de memória do dispositivo, em muitos casos seu uso pode ser interessante). Marque a opção *USE HOST GPU*, fazendo com que sejam utilizadas as bibliotecas do OpenGL do próprio computador, deixando a experiência com o emulador mais agradável/veloz. Para finalizar, clique em **OK**, como mostra a **Figura 2**.

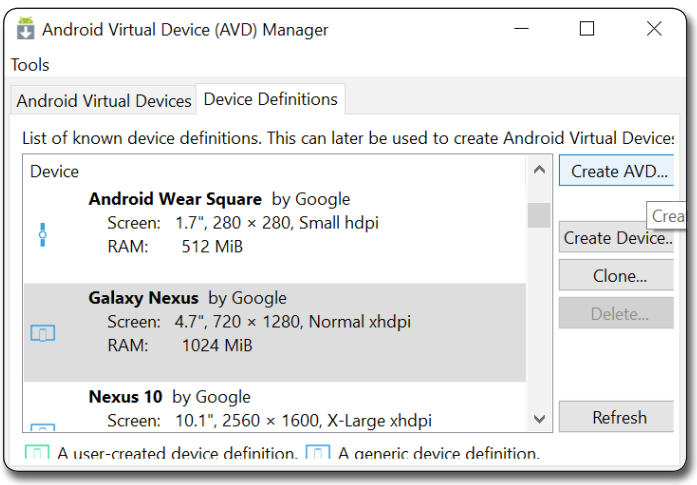

Figura 1. Configurando o emulador Android - Parte 1

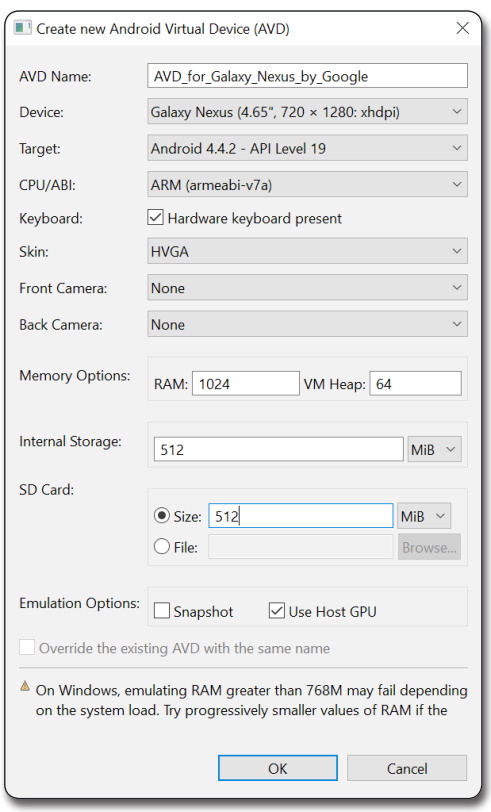

Figura 2. Configurando o emulador Android - Parte 2

Agora que o emulador está criado e configurado, na aba **Android Virtual Devices**, selecione-o e inicie-o (clique em **Start**) como mostra a **Figura 3**.

Nota

Lembre-se de manter o emulador ativado quando for testar suas aplicações Delphi Android.

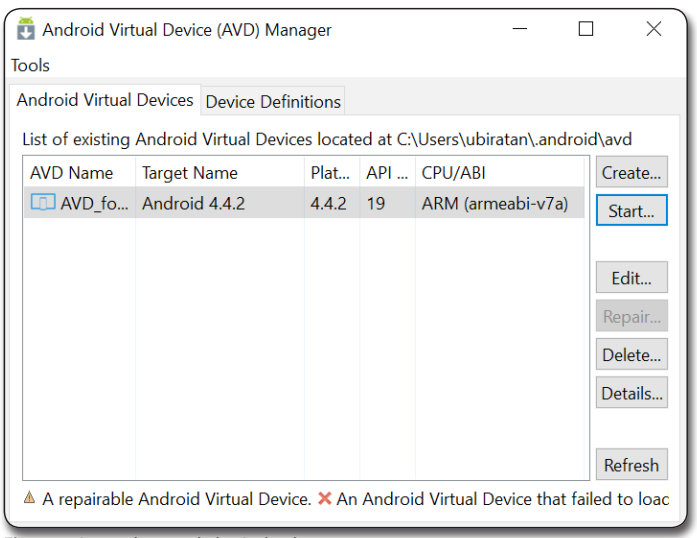

Figura 3. Iniciando o emulador Android

Construindo o leitor de assinaturas

Para realizar a implementação de nossa aplicação para dispositivos móveis basta iniciar um novo projeto e selecionar a opção "Multi-Device Application – Delphi". Será apresentada uma tela com alguns ***templates*** que podem facilitar o trabalho do desenvolvedor. No nosso caso, escolheremos o ***blank***.

Como a aplicação em questão está sendo desenvolvida para o ambiente ***Android***, no campo ***Style*** selecionaremos ***Android*** e no campo ***Views***, a opção ***Android 4" Phone*** (entretanto, sinta-se à vontade para escolher outras resoluções), assim como é apresentado na **Figura 4**.

Por fim, selecione o ambiente para o qual deseja desenvolver e o emulador/dispositivo onde irá realizar os testes. Veja o processo na **Figura 5**.

Reparando o erro com arquivo ZipAlign.exe

Existe uma grande possibilidade de, no momento da compilação do projeto no RAD Studio, ocorrer o seguinte erro: ***Could not find program 'C:\.....\sdk\tools\ZipAlign.exe'***. Caso se depare com esse problema, siga as instruções a seguir e compile seu sistema sem mais problemas:

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Busque pela pasta **Embarcadero studio XXX** e em seguida selecione o aplicativo **Android tools**. Na janela que será apresentada, selecione um ou todos os pacotes ***Android SKD Build-Tools*** (recomenda-se instalar o mais recente disponível), como mostra a **Figura 6**.

Ao terminar de baixar e instalar os pacotes, abra a pasta ***sdk\build-tools\23.0.1*** (note que 23.0.1 corresponde à versão do ***Android SDK Build-Tools*** instalada) e copie o arquivo ***ZipAlign.exe*** para a pasta ***sdk\tools***. As duas pastas citadas acima podem ser encontradas no seguinte endereço:

C:\Users\Public\Documents\Embarcadero\Studio\15.0\PlatformSDKs\adt-bundle-windows-x86-20131030

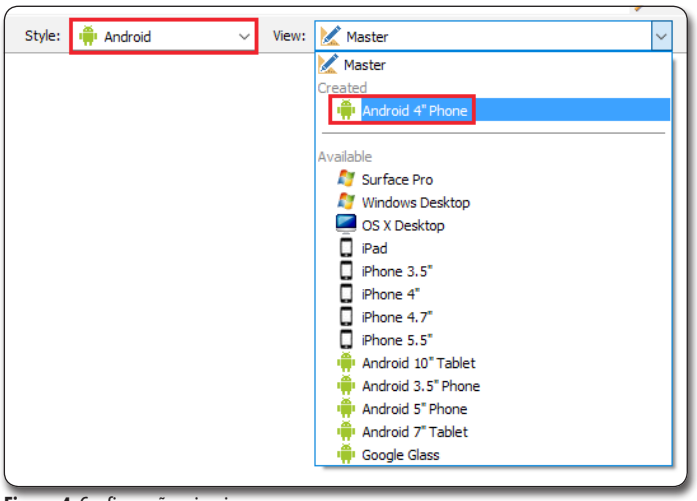

Figura 4. Configurações visuais

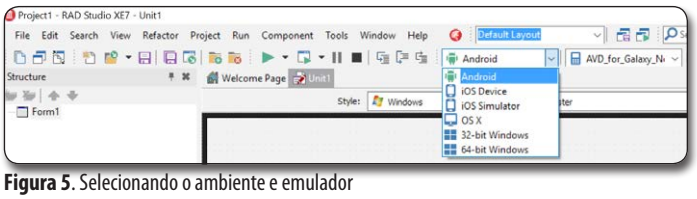

Figura 5. Selecionando o ambiente e emulador

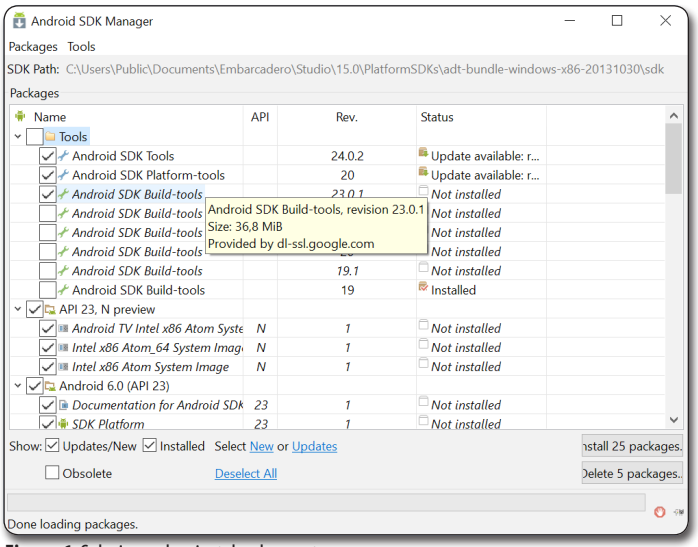

Figura 6. Selecionando e instalando pacotes

Com isso, o problema está resolvido e o programa compilará corretamente. Em alguns casos pode ser necessário reconfigurar o emulador.

Tela inicial – layout

O projeto será composto por uma tela inicial que nos permite acessar dois módulos, um responsável pelo recolhimento das assinaturas e outro, pelo envio das mesmas por e-mail. Para criarmos essa tela de acesso, basta adicionar no formulário criado automaticamente pelo RAD Studio os seguintes componentes: ***TLayout*** (muito importante, pois é o componente responsável pelo agrupamento dos demais componentes), ***TImage***, ***TLabel*** e dois componentes ***TButton***. Esses componentes, depois de alteradas suas propriedades, darão à aplicação uma aparência inicial semelhante à que pode ser observada na **Figura 7**.

Figura 7. Tela inicial do sistema

Tela inicial – implementação

Criada a tela inicial do sistema, daremos início às demais partes do mesmo: o módulo de recolhimento de assinaturas e o módulo responsável por enviar as assinaturas recolhidas por e-mail. Para isso criaremos um novo formulário seguindo a sequência *File > New > Multi-Device Form – Delphi*.

Na tela que será apresentada, selecione a opção ***HD FORM*** e clique em ***OK*** para retornar um formulário básico limpo, sem qualquer ***layout*** pré-definido. Repetiremos o processo indicado para criação de um terceiro formulário.

Na **Listagem 1**, é apresentada parte da implementação da classe ***uPrincipal*** (correspondente ao **Form1**). Vale ressaltar, neste ponto, que o código dessa listagem se refere somente aos eventos que providenciam a chamada para as telas recém-criadas, sendo a implementação completa dessa classe apresentada posteriormente.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 1. Chamadas para as telas de assinatura e envio de e-mail.

```
01. procedure TForm1.Button1Click(Sender: TObject);
02. begin
03.   if Form3 = nil then
04.    begin
05.      application.CreateForm(TForm3, Form3);
06.      Form3.Show;
07.      Form3 := nil;
08.    end;
09. end;
10.
11. procedure TForm1.Button2Click(Sender: TObject);
12. begin
13.   if Form2 = nil then
14.    begin
15.      application.CreateForm(TForm2,Form2);
16.      Form2.Show;
17.      Form2 := nil;
18.    end;
19. end;
```

Veja que só adicionamos uma chamada para o terceiro formulário no evento *Click* do *Button1* e para o segundo formulário no evento *Click* do *Button2*. Deve-se também, na seção *implementation*, adicionar a cláusula *uses* e referenciar as *units* dos dois formulários criados, como pode ser observado abaixo. Note que as unit1, unit2 e unit3 foram renomeadas, passando, respectivamente, para as seguintes nomenclaturas *uPrincipal*, *uEmail* e *uTelaAssinatura*:

```
1. implementation
2. uses uEmail, uTelaAssinatura;
```

A inserção das units *uEmail* e *uTelaAssinatura* após a cláusula *uses* permite que a unit que as referencia acesse seus métodos e propriedades públicas. Dessa forma, nossa tela principal, denominada *uPrincipal*, poderá invocar o método **show** das duas *units* referenciadas.

Deve-se também criar, na *uPrincipal*, a variável *caminhoAssinatura* (do tipo **string**), a função *get_caminhoAssinatura* (que retorna uma **string**) e um procedimento *set_caminhoAssinatura* (que recebe uma **string** como entrada). A implementação final dessa unit é semelhante à apresentada na **Listagem 2**.

Recolher assinatura – layout

Para o formulário *Form3* vamos inserir (nessa ordem) os seguintes componentes visuais para criar uma interface gráfica amigável ao usuário: *TLayout*, *TRoundRect*, *TLine*, *TText* e um *TImageControl*. Cada um desses sofrerá alterações e é importante que os mesmos sejam adicionados na ordem indicada. Além disso, sempre antes de adicionar o seguinte, realize as devidas alterações no componente então inserido.

Selecione o componente *TLayout* e mude sua propriedade *align* para *client* para que ele ocupe todo o espaço da tela. Agora, selecione o componente *TRoundRect* e defina sua propriedade *align* como *bottom*. Posteriormente, em sua propriedade *corners*, coloque a seguinte configuração:

Listagem 2. Implementação da uPrincipal.

```
01. unit uPrincipal;
02.
03. interface
04.
05. uses
06.   System.SysUtils, System.Types, System.UITypes, System.Classes, System.Variants,
07.   FMX.Types, FMX.Controls, FMX.Forms, FMX.Graphics, FMX.Dialogs, FMX.StdCtrls,
08.   FMX.Objects, FMX.Layouts;
09.
10. type
11.  TForm1 = class(TForm)
12.   Layout2: TLayout;
13.   Label1: TLabel;
14.   Image1: TImage;
15.   Button2: TButton;
16.   Button1: TButton;
17.   procedure Button2Click(Sender: TObject);
18.   procedure Button1Click(Sender: TObject);
19. private
20.   { Private declarations }
21.   caminhoAssinatura : String;
22. public
23.   { Public declarations }
24.   procedure set_caminhoAssinatura(valor:String);
25.   function get_caminhoAssinatura:String;
26.  end;
27.
28. var
29.  Form1: TForm1;
30.
31. implementation
32.
33. {$R *.fmx}
34. {$R *.NmXhdpiPh.fmx ANDROID}
35.
36. uses uEmail, uTelaAssinatura;
37.
38. procedure TForm1.Button1Click(Sender: TObject);
39. begin
40.   if Form3 = nil then
41.   begin
42.     application.CreateForm(TForm3,Form3);
43.     Form3.Show;
44.     Form3 := nil;
45.  end;
46. end;
47.
48. procedure TForm1.Button2Click(Sender: TObject);
49. begin
50.   if Form2 = nil then
51.   begin
52.     application.CreateForm(TForm2,Form2);
53.     Form2.Show;
54.     Form2 := nil;
55.   end;
56. end;
57.
58. function TForm1.get_caminhoAssinatura: String;
59. begin
60.   result := caminhoAssinatura;
61. end;
62.
63. procedure TForm1.set_caminhoAssinatura(valor: String);
64. begin
65.   caminhoAssinatura := valor;
66. end;
67.
68. end.
```

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

topleft = false; topright = true; bottomleft = false; bottomright = false.

Essas configurações darão o formato da imagem de base da assinatura, mantendo suas bordas superior esquerda, inferior esquerda e inferior direita com ângulo de 90º e a borda superior direita arredondada. Essas alterações são somente para manter uma boa apresentação visual.

Ainda nesse componente, insira o valor ***120***, ou outro a seu critério, em sua propriedade ***height***. Por fim, selecione a propriedade ***margins*** e dentro das subpropriedades ***bottom***, ***right*** e ***left*** insira o valor 8 para garantir que o componente mantenha uma certa distância entre suas margens e os limites da tela do dispositivo.

Selecionaremos, agora, o componente ***TLine*** e mudaremos as seguintes propriedades:

• ***align*** - definimos ***bottom*** para a linha da assinatura estar sempre no limite inferior do componente ***TRoundRect***;

• ***linetype*** - configuramos como ***top***, fazendo com que a linha da assinatura mantenha uma certa distância até a linha de base do componente ***TRoundRect***;

Depois, selecione o componente ***TText*** e marque a propriedade ***hitText*** como **false**, evitando que, sempre que o componente estiver em foco, sejam apresentadas mensagens ao usuário (balõezinhos informativos). Para que o texto e a linha que indica o local da assinatura fiquem dentro do ***TRoundRect***, primeiro insira esse e posteriormente insira os demais dentro dele.

Nota

A view Structure exibe a estrutura de componentes de um determinado form e permite que eles sejam reorganizados. Assim, se você inserir algum componente fora da ordem ou se a estrutura final dos componentes não representa o resultado imaginado, basta arrastar e soltar um componente para o local desejado.

Por fim, configuraremos o componente ***TImageControl*** para que fique bem posicionado na tela do dispositivo. O resultado final deve ser algo parecido com o que é apresentado na **Figura 8**.

Recolher assinatura – implementação

Após concluída a montagem da interface gráfica do leitor de assinaturas, realizaremos as devidas implementações para que o módulo tenha funcionalidade, mas antes é preciso entender seu funcionamento. O usuário realizará a assinatura diretamente no componente ***TRoundRect***, que é o componente retangular com uma das bordas arredondadas da **Figura 8**. Realizada a assinatura, quando o usuário clicar sobre o componente ***TImageControl***, a mesma será transferida para esse componente (o retângulo branco na parte superior da tela) e terá uma cópia salva na memória interna do dispositivo.

OnClick do ImageControl

Para garantirmos o funcionamento dessa última funcionalidade precisamos implementar as ações do componente ***TImageControl***. Selecione-o e em seu evento ***OnClick*** insira o código apresentado na **Listagem 3**.

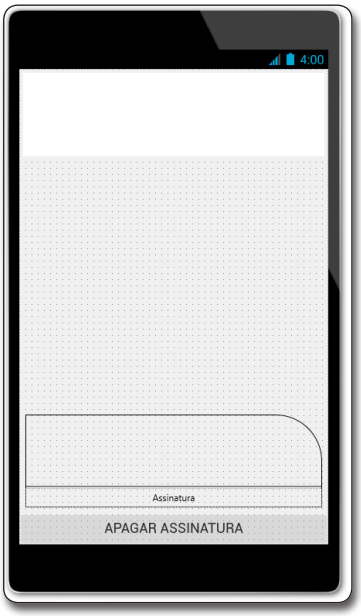

Figura 8. Layout da tela de recolhimento de assinaturas

Listagem 3. Implementação do evento OnClick do ImageControl.

```
01. procedure TForm3.ImageControl1Click(Sender: TObject);
02. var
03.   caminho : string;
04. begin
05. ImageControl1.Bitmap.Clear(TAlphaColors.White);
06. ImageControl1.Bitmap.SetSize(Round(ImageControl1.Width)
      Round(ImageControl1.Height));
07. caminho := TPath.GetPicturesPath+'/Test'+IntToStr
(Random(1000))+'.png';
08.
09. If RoundRect1.Canvas.BeginScene then
10. begin
11.   try
12.     ImageControl1.Bitmap.Canvas.BeginScene;
13.     RoundRect1.PaintTo(ImageControl1.Bitmap.Canvas, RectF
        (0,0,Round(ImageControl1.Width),Round(ImageControl1.Height)));
14.   finally
15.     RoundRect1.Canvas.EndScene;
16.     ImageControl1.Bitmap.Canvas.EndScene;
17.   end;
18. end;
19. ImageControl1.Bitmap.SaveToFile(caminho);
20. ImageControl1.LoadFromFile(caminho);
21. Form1.set_caminhoAssinatura(caminho);
22. end;
```

Na linha 3, tem-se a criação de uma variável **caminho**, do tipo **string**, que armazenará o caminho onde será salva a imagem da assinatura. Na linha 5, realiza-se a limpeza do componente ***ImageControl1***, garantindo que toda imagem nele armazenada seja apagada, e na linha 6 redimensiona-se o tamanho do bitmap que será criado (apesar de o objeto ser referenciado como bitmap, nossa imagem será arquivada como png, pois o Android, por padrão, não suporta bitmaps).

Na linha 7 recuperamos o caminho onde a imagem será armazenada utilizando o método ***GetPicturesPath***, que retorna a pasta

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

pública de armazenamento de imagens (no sistema Android) referente ao aplicativo em uso. Adiciona-se também à string o texto "Test", o resultado de um valor randômico de 0 a 1000, e a extensão ".png", formando-se, assim, o caminho completo da assinatura armazenada.

Na linha 9, verifica-se se foi possível iniciar uma escrita com a propriedade ***Canvas*** do objeto ***RoundRect1***. Se o retorno for verdadeiro, inicia-se uma escrita no componente ***ImageControl1***, como pode ser observado na linha 12. Na linha 13 temos o método ***PaintTo*** do objeto ***RoundRect***, que copia para o componente ***ImageControl1*** o conteúdo do componente ***RoundRect1***, iniciando das posições x=0 e y=0 e indo até as dimensões do componente ***ImageControl1***. Nas linhas 15 e 16 finaliza-se as escritas nos componentes ***ImageControl1*** e ***RoundRect1***.

Por fim, armazena-se a imagem presente no componente ***ImageControl1*** no caminho armazenado na variável ***caminho***, realiza-se (na linha 20) a leitura dessa imagem para que a mesma seja aberta no componente ***ImageControl1*** e preenche-se a variável ***caminhoAssinatura*** da ***uPrincial*** com o valor armazenado na variável ***caminho*** através do método ***set_caminhoAssinatura***.

Existe uma forma bem simples de deixamos nossa imagem com um aspecto mais agradável, evitando o fundo cinza. Para isso precisamos selecionar o componente ***RoundRect1*** e procurar pela propriedade ***Fill***, que possui uma subpropriedade denominada ***Kind***. Selecione-a e altere seu valor para ***bkNone***, assim garantindo que o fundo apresentado na imagem não seja mais cinza, mas sim transparente, ou, pelo padrão, branco.

Nota

Para os que utilizarão essa ferramenta com o propósito de recolher assinaturas para contratos, recomenda-se que o melhor seria inserir uma imagem de fundo que simule esse contrato, ou mesmo adicionar o componente ImageControl1 em um objeto gerador de relatórios, para então criar um pdf do mesmo e anexá-lo ao e-mail. Dessa forma, a ferramenta terá aparência e recursos mais profissionais.

Ainda no ***TImageControl*** selecione a propriedade ***EnableOpenDialog*** e marque-a como **false**, impedindo que sempre que for clicado no componente seja aberta uma janela solicitando um local para armazenamento da imagem.

Escrevendo a assinatura no TRoundRect

Ainda no ***Form3***, em sua seção **uses**, insira a biblioteca ***System.Generics.Collections***, pois entre os vários recursos dessa biblioteca está a definição do tipo ***TList***. Esse é um tipo abstrato de dados que utilizaremos juntamente com um registro, que está apresentado na **Listagem 4**.

Veja que o código cria, através da *cláusula* ***Type***, um novo tipo abstrato de dados para o projeto, nomeado ***TSigCapRec***. Ele é um registro capaz de guardar dois tipos distintos de dados: um ***TPointF*** e um ***Byte***. Agora, para dar continuidade ao processo, serão criadas duas variáveis e uma procedure na seção ***public*** do projeto, como mostra o código da **Listagem 5**.

Listagem 4. Implementação do Record.

```
01. type
02.   TSigCapRec = Record
03.     CurPos: TPointF;
04.     PosState: Byte;
05.   end;
```

Listagem 5. Variáveis e procedimentos da seção public.

```
01. public
02.   Signature: Tlist<TSigCapRec>;
03.   AndroidButton:Boolean;
04.   procedure AddPoint(const aX, aY: single; const aState:Byte);
```

A seção public do ***Form3*** contém uma variável denominada ***Signature***, do tipo ***TList*** de ***TSigCapRec***, ou seja, uma lista de dados do tipo ***TSigCapRec***. Essa variável armazenará cada um dos pontos da assinatura realizada pelo usuário, e em cada novo ponto adicionado será informado também o estado da assinatura (0 – marca o início ou retomada da escrita de uma assinatura, 1 – escrita em curso, 2 – encerramento de uma parte ou de toda a escrita — veja o **BOX 1**). Cria-se também a variável ***AndroidButton***, que auxiliará o sistema na tarefa de controlar se uma parte da assinatura, ou toda ela, já se encontra finalizada ou não. Por fim, é definido o método ***AddPoint***, que se encarregará de preencher a lista anteriormente criada, registrando os pontos marcados pelo usuário. A implementação da procedure ***AddPoint*** pode ser observada na **Listagem 6**.

Listagem 6. Implementação do procedimento AddPoint.

```
01. procedure TForm3.AddPoint(const aX, aY: Single; const aState: Byte);
02.   var P: TSigCapRec;
03. begin
04.   P.CurPos:=PointF(aX,aY);
05.   P.PosState:=aState;
06.   If Signature.Count-1 < 0 then P.PosState := 0;
07.   If P.PosSTate <> 1 then
08.     Signature.Add(P)
09.   else if P.CurPos.Distance(Signature.Last.CurPos) > 2.0 then
10.     Signature.Add(P);
11.   RoundRect1.Repaint;
12. end;
```

O procedimento recebe um ponto percorrido pelo cursor e o estado atual, e os armazena no registro, como pode ser observado nas linhas 4 e 5. Quando o usuário inicia a escrita, a lista ***Signature*** está vazia, logo ele armazena o estado 0 na variável ***PosState*** conforme a linha 6, indicando ao sistema que a escrita foi iniciada. Na linha 7, verifica-se se o estado é diferente de 1: se for, insere-se um novo ponto na lista ***Signature***. Caso o estado seja igual a 1, o ponto só será inserido na lista se a distância em relação ao ponto anterior for superior a 2 pixels, o que evita inserção de pontos desnecessariamente.

Sempre que um novo ponto é registrado é necessária a execução do evento ***Repaint*** do componente ***RoundRect1***. Esse evento atualiza o componente com os novos pontos que foram adicionados

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

à lista de pontos da assinatura, procedimento esse ocorrido na linha 11 da **Listagem 6**.

A próxima atividade a ser realizada é a inicialização de algumas das variáveis e componentes utilizados no ***Form3***. Para tanto, selecione esse formulário e, em seu evento ***onCreate***, insira o código demonstrado na **Listagem 7**. No código, somente quatro ações são realizadas: a inicialização da lista (linha 3), a inclusão de um texto no componente ***Text1*** (linha 4), redefinição do tamanho do bitmap do componente ***TImageControl1***, e a limpeza desse mesmo bitmap, acrescida da propriedade *TAlphaColors.White*, garantindo que o mesmo tenha o branco como cor de fundo padrão.

Agora, vamos inserir o código apresentado na **Listagem 8** no evento ***OnDestroy*** do ***Form3*** para liberar a memória ocupada pela lista (linha 3).

Cuidaremos, então, do evento ***OnMouseMove*** do componente ***TRoundRect***, onde implementaremos o código responsável por desenhar a assinatura do usuário, como mostra a **Listagem 9**.

O método ***MouseMove*** do componente ***TRoundRect*** verifica se o ponteiro do mouse está em movimento com seu botão de ação pressionado (botão esquerdo). Contudo, como estamos desenvolvendo um sistema ***touchScreen***, essa verificação baseia-se em outro parâmetro: é verificado se existe contato contínuo do dispositivo de escrita (dedo ou outro dispositivo utilizado para interação) com a tela do aparelho. Caso exista, o valor da variável ***AndroidButton*** é verificado a fim de indicar ao sistema se a assinatura em questão se trata da continuidade de uma assinatura em curso (nesse caso ***AndroidButton*** é ***true*** e chamamos o ***AddPoint*** passando o estado ***1***) ou não. Se o valor for ***false***, estamos iniciando uma assinatura ou retomando a escrita de uma assinatura depois de um espaço em branco, por exemplo. Portanto, o método ***AddPoint*** receberá o estado **0**.

Listagem 7. Implementação do FormCreate.

```
1.  procedure TForm3.FormCreate(Sender: TObject);
2.  begin
3.     Signature:=TList<TSigCapRec>.Create;
4.     Text1.Text:= 'Assinatura '+DateToStr(now);
5.     ImageControl1.Bitmap.SetSize(Round(ImageControl1.Width),
       Round(ImageControl1.Height));
6.     ImageControl1.Bitmap.Clear(TAlphaColors.White);
7.  end;
```

Listagem 8. Implementação do FormDestroy.

```
1.  procedure TForm3.FormDestroy(Sender: TObject);
2.  begin
3.     FreeAndNil(Signature);
4.  end;
```

Listagem 9. Implementação do OnMouseMove do componente TRoundRect1.

```
1.  procedure TForm3.RoundRect1MouseMove(Sender: TObject;
    Shift: TShiftState;  X, Y: Single);
2.  begin
3.     if ssLeft in Shift then
4.        If Not AndroidButton then
5.        begin
6.           AddPoint(X,Y,0);
7.           AndroidButton:=true;
8.        end
9.        else
10.          AddPoint(X,Y,1);
11. end;
```

Ainda no componente ***RoundRect1***, deve-se implementar o código da **Listagem 10** no seu evento ***OnMouseUp***. Nele, temos comandos que permitem ao sistema identificar quando a assinatura (ou o trecho da assinatura em questão) foi encerrada.

Nesse código setamos a variável ***AndroidButton*** como ***false*** e enviamos ao procedimento ***AddPoint*** as posições x e y atuais e o estado **2**, indicando que o usuário encerrou a escrita da assinatura ou de um trecho da mesma.

Listagem 10. Implementação do OnMouseUp do componente TRoundRect1.

```
1.  procedure TForm3.RoundRect1MouseUp(Sender: TObject;
    Button: TMouseButton; Shift: TShiftState;  X, Y: Single);
2.  begin
3.     AndroidButton := false;
4.     AddPoint(X,Y,2);
5.  end;
```

Seguiremos agora para a implementação do método ***Paint*** do componente ***RoundRect1***, que é chamado sempre que o componente é impresso em tela. Devemos implementar nesse método o código responsável por desenhar as linhas que acompanham os pontos percorridos pelo usuário durante o processo de assinatura. O código responsável por realizar essa operação pode ser observado na **Listagem 11**.

Na linha 6 o sistema verifica se existe algum ponto de assinatura registrado na lista: caso não exista a função é interrompida, evitando processamentos desnecessários. Das linhas 7 a 9 são realizadas somente algumas configurações relacionadas ao objeto ***Canvas***, encarregado de representar graficamente os pontos por onde o usuário percorreu com seu toque, formando assim sua assinatura. Já na linha 10 temos um laço de repetição do tipo ***for***, que percorre toda a lista de assinaturas (***Signature***), e, na linha 11, inicia-se uma estrutura de decisão do tipo ***case***, onde são testados os seguintes casos:

- Linha 12: sendo o estado do cursor **0**, então a variável ***P1*** recebe a posição atual do cursor e nenhuma outra operação é realizada, pois se entende que o usuário está iniciando ou retomando sua escrita;
- Linha 13: caso o estado seja **1**, então o sistema entende que o usuário está em processo de assinatura. Assim sendo, a posição atual do cursor é enviada para ***P2*** (linha 14) e logo em seguida é executado o evento ***DrawLine*** do objeto ***Canvas***, recebendo como parâmetros a posição inicial do cursor, armazenada na variável ***P1***, a posição atual do cursor, que se encontra armazenada em ***P2***, e o tipo de linha que vai ser desenhada, definida na propriedade ***Canvas.Stroke*** (que se refere a uma linha grossa).
- Linha 18: caso o estado seja **2**, realiza-se exatamente o mesmo procedimento descrito no bullet anterior, mas não é mais necessário atribuir a posição atual do cursor à variável ***P1***, pois não será traçada nenhuma linha partindo desse ponto já que o estado **2** caracteriza finalização.

Essas operações são o suficiente para que possamos ter algo semelhante ao que é apresentado na **Figura 9**.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 11. Implementação do OnPaint do componente TRoundRect1.

```
01.  procedure TForm3.RoundRect1Paint(Sender: TObject; Canvas: TCanvas;
     const ARect: TRectF);
02.  var
03.    P:TSigCapRec;
04.    P1,P2: TPointF;
05.  begin
06.    if Not (Signature.Count-1 > 0) then Exit;
07.    Canvas.Stroke.Dash:=TStrokeDash.sdSolid;
08.    Canvas.Stroke.Thickness:=2;
09.    Canvas.Stroke.Color:=TAlphaColorRec.Darkblue;
10.    for P in Signature do begin
11.      case P.PosState of
12.        0: P1:=P.CurPos;
13.        1: begin
14.             P2:=P.CurPos;
15.             Canvas.DrawLine(P1,P2,1,Canvas.Stroke);
16.             P1:=P.CurPos;
17.           end;
18.        2: begin
19.             P2:=P.CurPos;
20.             Canvas.DrawLine(P1,P2,1,Canvas.Stroke);
21.           end
22.      end;
23.      RoundRect1.Repaint;
24.  end;
```

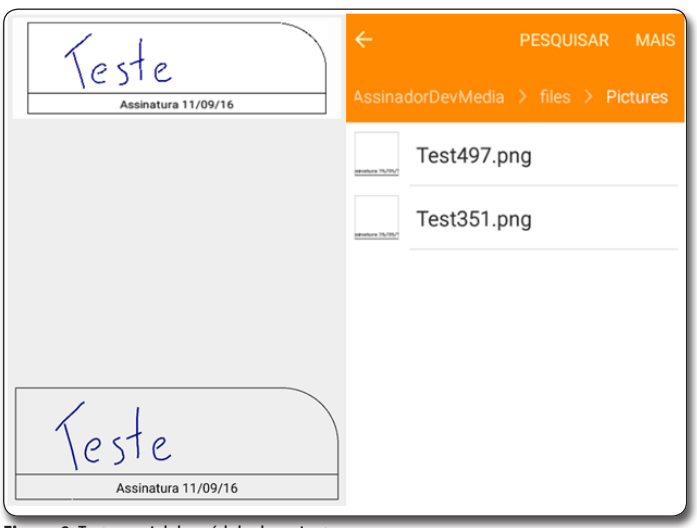

Figura 9. Teste parcial do módulo de assinatura

Note que na parte direita da figura temos algumas imagens que foram criadas no processo de assinatura, nomeadas com o prefixo "Test" e um valor randômico, como explicado no tópico anterior.

Limpando o TRoundRect

Esse projeto ainda não está concluído, pois não foi considerada a possibilidade de falha. Supondo que o usuário, por algum motivo, tenha errado sua assinatura, como poderia esse apagar a mesma?

Para garantir que o usuário tenha a opção de realizar uma nova assinatura caso o mesmo tenha cometido algum erro na escrita,

Listagem 12. Apagando a assinatura.

```
1. procedure TForm3.Button1Click(Sender: TObject);
2.begin
3.  Signature.Clear;
4.  RoundRect1.Repaint;
5.end;
```

BOX 1. Estados da nossa aplicação

Ao iniciarmos nossa assinatura, o estado para aplicação é 0, mas assim que começamos a escrever, o estado muda para 1, o que caracteriza que a escrita está em curso.

Em uma assinatura podemos ter alguns espaços em branco, como a separação entre o nome e o sobrenome, um acento etc. Em nossa aplicação, quando um caso assim acorre ou quando finalizamos completamente a assinatura, deixa de existir o contato do dedo do usuário com o componente TRoundRect, o que leva o estado a mudar para 2. Se desejarmos continuar escrevendo, a aplicação tratará essa retomada de forma bem parecida com o início da escrita, isto é, o estado muda novamente para 0. A partir daí o processo se repete até a finalização definitiva.

insira um novo botão na tela, defina sua propriedade ***Text*** como "Apagar Assinatura", e, clicando duas vezes sobre o mesmo, insira o código da **Listagem 12** em seu método de clique.

Nesse caso, limpa-se todos os pontos armazenados na variável ***Signature*** (linha 3) e, posteriormente, repinta-se o componente ***TRoundRect1*** (linha 4) para que as mudanças tenham efeito.

Envio do e-mail - layout

Iniciaremos criando a interface gráfica do módulo. Para isso, inserimos no ***Form2*** os seguintes componentes: ***TLayout***, ***TButton***, ***TEdit*** e ***TImage***. O componente ***TLayout*** receberá em sua propriedade ***Align*** o valor ***Client***, assim como feito nas outras telas, e agrupará os demais componentes. O ***TEdit***, por sua vez, receberá o valor ***Top*** em sua propriedade ***Align*** e ***EmailAddress*** em ***KeyboardType***. Dessa forma, nosso campo de texto será configurado para receber endereços de e-mail.

Em seguida, selecione o componente ***TButton***, configure sua propriedade ***Align*** como ***Bottom***, e ***Text*** como "Enviar". Por fim, vamos configurar o componente ***TImage***, colocando na sua propriedade ***Align*** o valor ***Client*** e selecionando uma imagem que represente de forma clara e simplificada seu projeto. Pronto, a tela resultante desse processo já pode ser observada na **Figura 10**.

Basicamente, o que esse módulo fará é enviar um e-mail com um anexo para o destinatário que será inserido no ***Edit*** utilizando os próprios recursos já disponíveis no ***Google Android***, o que garante que o mesmo seja compatível com qualquer versão desse sistema.

Envio do e-mail - implementação

Primeiramente vamos inserir alguns ***namespaces*** (bibliotecas) na cláusula ***uses*** da ***uEmail***: *FMX.Helpers.Android*, *Androidapi.Jni.GraphicsContentViewText*, *Androidapi.Jni.Net*, *Androidapi.Jni.JavaTypes* e *Androidapi.JNIBridge*. Todos são nativos do próprio sistema operacional ***Google Android*** e vão garantir o bom funcionamento de nosso aplicativo. Assim sendo, as primeiras linhas de nossa ***unit*** devem ficar semelhante ao código apresentado na **Listagem 13**.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 13. Linhas iniciais da unit de envio de e-mail.

```
01.  unit uEmail;
02.
03.  interface
04.
05.  uses
06.  System.SysUtils, System.Types, System.UITypes, System.Classes,
     System.Variants,
07.  Androidapi.helpers, FMX.Types, FMX.Controls, FMX.Forms,
     FMX.Graphics, FMX.Dialogs,
08.  FMX.Objects, FMX.StdCtrls, FMX.Controls.Presentation, FMX.Edit,
09.  Androidapi.Jni.GraphicsContentViewText, Androidapi.Jni.Net,
10.  Androidapi.Jni.JavaTypes, Androidapi.JNIBridge, FMX.Helpers.Android,
11.  FMX.Platform.Android, FMX.MobilePreview, AndroidAPI.jni.OS,
     AndroidApi.Jni.App,
12.  FMX.Layouts;
13.
14.  type
15.    TForm2 = class(TForm)
16.    Layout1: TLayout
17.    Edit1: TEdit;
18.    Button1: TButton;
19.    Image1: TImage;
20.    procedure Button1Click(Sender: TObject);
21.    private
22.    { Private declarations }
23.    public
24.    { Public declarations }
25.  end;
26.
27.  var
28.    Form2: TForm2;
29.
30.  implementation
31.
32.  {$R *.fmx}
33.  {$R *.NmXhdpiPh.fmx ANDROID}
34.
35.  uses
36.    uPrincipal;
```

Figura 10. Tela do módulo de envio de e-mail

Agora nosso módulo tem tudo o que precisa para funcionar corretamente, entretanto é preciso ainda implementar sua principal funcionalidade, que realizará a integração desse com a ferramenta gerenciadora de e-mails de escolha do usuário. Não limite o usuário quanto a escolha da ferramenta: manter uma ferramenta específica é "engessar" a aplicação, obrigando o usuário a aceitar uma escolha feita pelo desenvolvedor e não por ele.

Para isso, iremos realizar algumas comunicações diretamente com o sistema operacional através da utilização de ***Intents***. ***Intents*** são classes do sistema operacional Android (***android.content.Intent***) que permitem aplicações de terceiros enviarem chamadas/mensagens para o sistema operacional a fim de realizar alguma atividade. Essa classe define padrões e interfaces que são interpretadas pelo sistema operacional, permitindo que o mesmo realize o que foi solicitado. É importante citar, também, que para cada operação existe um ***Intent***.

A funcionalidade da tela é codificada, de fato, no evento ***OnClick*** do componente ***Button1***, conforme apresentado na **Listagem 14**.

Listagem 14. Implementação do evento OnClick do botão de envio de e-mail.

```
01.  procedure TForm2.Button1Click(Sender: TObject);
02.  var
03.    Destinatario : string;
04.    Intent : JIntent;
05.    Destinatarios: TJavaObjectArray<JString>;
06.    Uri: Jnet_Uri;
07.    AttachmentFile: JFile;
08.    Anexo : string;
09.  begin
10.  if Edit1.Text <> EmptyStr then
11.    begin
12.      Destinatario        := Edit1.Text;
13.      Destinatarios       := TJavaObjectArray<JString>.Create(1);
14.      Anexo := Form1.get_caminhoAssinatura;
15.
16.      Intent := TJIntent.JavaClass.init(TJIntent.JavaClass.ACTION_SEND);
17.      Intent.setFlags(TJIntent.JavaClass.FLAG_ACTIVITY_NEW_TASK);
18.      Destinatarios.Items[0] := StringToJString(Destinatario);
19.      Intent.putExtra(TJIntent.JavaClass.EXTRA_EMAIL, Destinatarios);
20.      Intent.putExtra(TJIntent.JavaClass.EXTRA_SUBJECT, StringToJString
         ('E-mail de Teste'));
21.      Intent.putExtra(TJIntent.JavaClass.EXTRA_TEXT, StringToJString
          ('corpo do email'));
22.      Intent.setType(StringToJString('vnd.android.cursor.dir/email'));
23.      if Anexo <> '' then
24.      begin
25.        AttachmentFile := TJFile.JavaClass.init(StringToJString(Anexo));
26.        Uri := TJnet_Uri.JavaClass.fromFile(AttachmentFile);
27.        Intent.putExtra(TJIntent.JavaClass.EXTRA_STREAM, TJParcelable.Wrap
           ((Uri as ILocalObject).GetObjectID));
28.      end;
29.      SharedActivity.startActivity(TJIntent.JavaClass.createChooser
         (Intent,StrToJCharSequence('Qual aplicativo deseja usar?')));
30.      end;
31.  end;
```


DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Na linha 4 é criada uma variável do tipo **JIntent**, que permitirá interagir com os recursos do sistema Android. Já na linha 5 temos a criação de uma variável do tipo ***TJavaObjectArray<JString>***, que será utilizada para armazenar um vetor de objetos **JString**, que é um tipo de dado utilizado no ***Android***.

Nas linhas 6 e 7 as variáveis ***Uri***, do tipo **Jnet_Uri**, e **AttachmentFile**, do tipo **JFile**, são criadas, as quais serão utilizadas para anexar arquivos ao e-mail enviado. Na linha 14 o valor da variável **caminhoAssinatura**, preenchido na tela de captura da assinatura, é recuperado através do método **get_caminhoAssinatura** e armazenado na variável **anexo**.

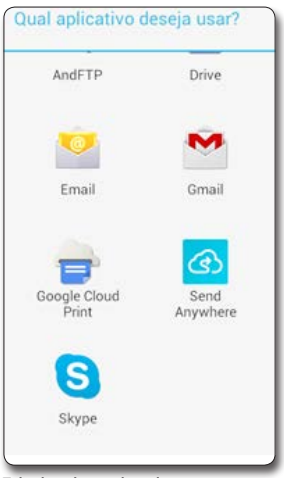

Figura 11. Tela de seleção da aplicação para envio do e-mail

Nota

Como toda implementação é feita em um sistema único, torna-se fácil criar um módulo de persistência que, após armazenar a imagem da assinatura em disco, guarde seu caminho para inserção automática como anexo no módulo de envio de e-mail.

Das linhas 19 a 22, várias invocações ao método ***putExtra*** do objeto ***Intent*** são feitas, o que permitirá inserir diversas informações no e-mail que desejamos enviar, como:

- linha 19 - destinatário;
- linha 20 - assunto;
- linha 21 - texto do corpo do e-mail;
- linha 22 - tipo de serviço do sistema Android que será utilizado.

Nas linhas 23 a 28, o sistema verifica a existência ou não de arquivos para anexar ao e-mail. Se existir algum, o mesmo é anexado. Ao ser executado, o módulo deve apresentar, após o preenchimento do campo do destinatário e o clique no botão "Enviar", uma tela semelhante à apresentada na **Figura 11**.

Repare que a junção de duas simples miniaplicações nos permite criar um sistema rápido e que oferece aos usuários recursos capazes de lhes trazer grande comodidade e segurança em suas aplicações. Pudemos também conhecer recursos relacionados à programação para ***Android*** utilizando o **RAD Studio**, ferramenta que oferece uma série de vantagens e facilidades para os desenvolvedores.

Autor

Ubiratan Roberte Cardoso Passos
ubiratanroberte@hotmail.com / upassos@gmail.com
Bacharel em Sistemas de Informação; Especialista em Produção de Software Livre; Mestre em Inteligência Computacional; Desenvolvedor Desktop, Web e Mobile; Gestor de Sistemas e Professor do Instituto Federal do Espírito Santo – Campus Alegre (ES)

Links:

RAD Studio - Embarcadero
https://www.embarcadero.com/br/products/rad-studio

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

FireMonkey e FireDAC: Construindo uma aplicação completa – Parte 1

Aprenda a construir uma aplicação completa de ordens de serviço

Este artigo faz parte de um curso

O Delphi sempre foi considerado uma das melhores IDEs para o desenvolvimento de aplicações desktop, possuindo uma gama de recursos que agilizam muito o trabalho dos programadores e contribuem para o rápido desenvolvimento de softwares comerciais. Nesse contexto, o VCL é o conjunto de componentes mais utilizado para a programação de softwares nesse estilo, sendo compatível somente com o Windows. Porém, como a utilização de diferentes sistemas operacionais (como OS X, Android e iOS) cresceu muito nos últimos anos, o Delphi passou por uma evolução natural, e foi criado o framework FireMonkey, que é compatível com todos esses sistemas operacionais, inclusive com o Windows.

Portanto, uma vantagem do FireMonkey sobre o VCL é que ele permite que um software seja compilado com código nativo para múltiplos dispositivos sem a necessidade de alterações no código fonte, desde que esse seja criado com componentes compatíveis. Dessa forma, esse framework oferece muitas vantagens para os desenvolvedores, porém existem algumas diferenças com relação a alguns componentes, como a ausência do DBGrid, que é um dos componentes mais utilizados para a listagem de registros. Esse fato acaba gerando um problema no processo de adaptação que os desenvolvedores precisam passar para começarem a utilizar o FireMonkey ao invés do VCL (quando aplicações multidispositivos são necessárias).

Tendo essa necessidade como base, o objetivo deste artigo é apresentar uma visão prática de como construir uma aplicação completa passo a passo utilizando o framework FireMonkey, o Delphi 10.1 Berlin e a engine FireDAC de acesso a dados. Nesta primeira parte serão mostrados os processos para a criação da base de dados de um cenário de ordens de serviços de um auto center, a definição do menu principal com acesso às janelas do

Fique por Dentro

Atualmente o Delphi dispõe de dois frameworks para o desenvolvimento visual de aplicações: o VCL (Visual Component Library) e o FireMonkey. O primeiro é muito conhecido pelos desenvolvedores desktop, pois acompanha o Delphi desde as primeiras versões, e é caracterizado por funcionar somente no Windows. Por outro lado, o FireMonkey é um framework mais recente, e sua característica principal é gerar aplicações nativas tanto para Windows quanto para OS X, Android e iOS. Dessa forma, é importante o conhecimento sobre como utilizar os componentes desse framework para construir aplicações também com características desktop, porém que possam ser executadas em um número maior de dispositivos.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

sistema e, por fim, o desenvolvimento de uma tela padrão de pesquisa, que será utilizada como janela base para as outras telas (utilizando herança). As próximas seções mostram o desenvolvimento completo do software.

Entendendo a base de dados

A **Figura 1** apresenta o modelo entidade relacionamento da base de dados, a qual é composta por seis tabelas. Na parte central do diagrama, encontra-se a tabela *m_ordem_servico*, a qual armazenará as principais informações sobre o cenário proposto. Para isso, ela se liga por meio de chaves estrangeiras às tabelas de clientes, formas de pagamento e veículos, ou seja, quando uma ordem de serviço for preenchida pelo usuário o mesmo terá que informar esses campos.

Considerando o cenário de auto center, quando um veículo é enviado para manutenção vários serviços podem ser realizados, como geometria, balanceamento, rodízio de pneus, lavação, entre vários outros. Dessa forma, uma ordem de serviço pode possuir vários serviços, enquanto que um mesmo tipo de serviço pode ser executado em várias ordens de serviço. Esse raciocínio nos leva a conclusão que esse relacionamento é do tipo muitos para muitos (N-N), o que requer que seja criada uma nova tabela, comumente chamada de entidade associativa, a qual fará a ligação entre as tabelas *m_ordem_servico* e *c_servico*. Esse relacionamento é demonstrado pela tabela *m_ordem_servico_servico*, que possui duas chaves estrangeiras: uma relacionando com a tabela *m_ordem_servico* e outra com a tabela *c_servico*. Em outras palavras, esse vínculo entre as tabelas caracteriza o recurso de mestre-detalhe, que é muito comum e de extrema importância para qualquer sistema comercial.

É importante enfatizar que a tabela *c_servico* armazenará o cadastro de todos os serviços ofertados pela loja, enquanto que a entidade associativa tem a função de registrar somente os serviços relacionados a uma ordem de serviço.

Criando a base de dados

Após o entendimento da estrutura da base de dados, o próximo passo é efetivamente criá-la no MySQL por meio da linguagem SQL. A **Listagem 1** apresenta os scripts para a definição das tabelas de cadastro, as quais possuem o prefixo "c" no início do nome. Na linha 01 a base de dados é criada, com o nome *ordem_servico*, enquanto que o restante do código apresenta a definição das tabelas de formas de pagamento (linhas 03 até 08), serviços (linhas 09 até 14), veículos (linhas 15 até 21) e clientes (linhas 22 até 30).

É possível observar que todas as tabelas possuem uma chave primária do tipo autoincremento, ou seja, os identificadores serão gerados automaticamente pelo MySQL. A engine de acesso a dados que usaremos será o FireDAC, o qual já possui recursos nativos para identificar campos do tipo autoincremento automaticamente independente do sistema gerenciador de banco de dados (consulte a seção **Links** para mais detalhes).

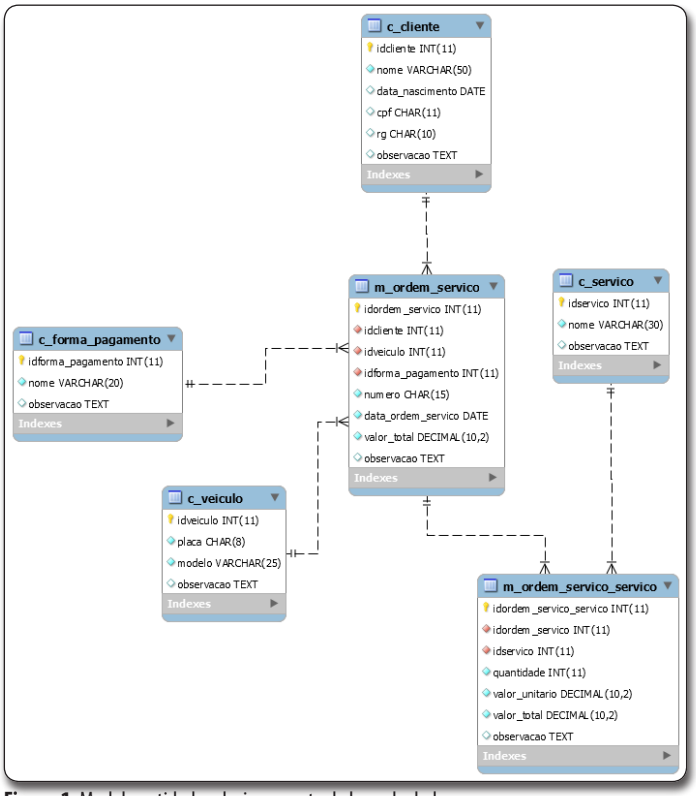

Figura 1. Modelo entidade relacionamento da base de dados

Listagem 1. Criação das tabelas de cadastro.

```
01 create database ordem_servico;
02 use ordem_servico;
03 create table c_forma_pagamento (
04  idforma_pagamento int not null auto_increment,
05  nome varchar(20) not null,
06  observacao text,
07  constraint pk_fp_idforma_pagamento primary key (idforma_pagamento)
08 );
09 create table c_servico (
10  idservico int not null auto_increment,
11  nome varchar(30) not null,
12  observacao text,
13  constraint pk_sv_idservico primary key (idservico)
14 );
15 create table c_veiculo (
16  idveiculo int not null auto_increment,
17  placa char(8) not null,
18  modelo varchar(25) not null,
19  observacao text,
20  constraint pk_vc_idveiculo primary key (idveiculo)
21 );
22 create table c_cliente (
23  idcliente int not null auto_increment,
24  nome varchar(50) not null,
25  data_nascimento date,
26  cpf char(11),
27  rg char(10),
28  observacao text,
29  constraint pk_cl_idcliente primary key (idcliente)
30 );
```

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

A **Listagem 2**, por sua vez, mostra os scripts para a inserção de alguns registros nas tabelas de cadastro, que serão utilizados para teste posteriormente. Conforme pode ser observado na listagem, são incluídos três registros para cada uma das quatro tabelas.

Listagem 2. Inserção de dados nas tabelas de cadastro.

```
01 insert into c_forma_pagamento (nome) values ('À vista');
02 insert into c_forma_pagamento (nome) values ('Boleto');
03 insert into c_forma_pagamento (nome) values ('Cartão');
04 insert into c_servico (nome) values ('Balanceamento');
05 insert into c_servico (nome) values ('Geometria');
06 insert into c_servico (nome) values ('Ajuste no freio');
07 insert into c_veiculo (placa, modelo) values ('AAA-8945', 'Fusion');
08 insert into c_veiculo (placa, modelo) values ('BBB-7451', 'Passat');
09 insert into c_veiculo (placa, modelo) values ('CCC-1254', 'Jetta');
10 insert into c_cliente (nome, data_nascimento, cpf, rg)
    values ('Jones', '1983-07-15', '11111111111', '1111111');
11 insert into c_cliente (nome, data_nascimento, cpf, rg)
    values ('Adaiane', '1985-11-11', '22222222222', '222222');
12 insert into c_cliente (nome, data_nascimento, cpf, rg)
    values ('Ana', '1965-04-21', '33333333333', '3333333');
```

A **Listagem 3** apresenta os scripts para a criação das duas tabelas de movimentação, ou seja, a das ordens de serviço (linhas 01 até 14) e a dos serviços relacionados (linhas 15 até 26). Na primeira tabela pode-se notar a existência de três chaves estrangeiras (linhas 11, 12 e 13), as quais estão vinculadas às tabelas de clientes, veículos e formas de pagamento. O mesmo acontece com a tabela de serviços da ordem de serviço, que possui uma chave estrangeira que a vincula com a ordem de serviço (relacionamento N-N mestre-detalhe — linha 24) e outra vinculando-a com a tabela de cadastro de serviços (linha 25).

Note que, diferentemente das tabelas de cadastro, os nomes das tabelas de movimentação recebem o prefixo "m". Essa diferenciação das tabelas por meio do prefixo pode ser considerada uma boa prática para diferenciação da função de cada uma delas, ou seja, enquanto as tabelas de cadastro armazenam os dados básicos do software, as tabelas de movimentação efetivamente apresentam os dados transacionais da empresa.

A **Listagem 4** mostra os scripts para inserção de alguns registros para teste nas duas tabelas de movimentação. Entre as linhas 01 e 03 são inseridas três ordens de serviço, enquanto que no restante da listagem encontra-se o código para inserir os respectivos serviços. Na linha 04 um serviço é inserido para a primeira ordem (*idordem_servico = 1*), nas linhas 05 e 06 são adicionados dois serviços para a segunda (*idordem_servico = 2*) e, por fim, na linha 07, um serviço é incluído para a terceira ordem de serviço (*idordem_servico = 3*).

Listagem 3. Criação das tabelas de movimentação.

```
01 create table m_ordem_servico (
02  idordem_servico int not null auto_increment,
03  idcliente int not null,
04  idveiculo int not null,
05  idforma_pagamento int not null,
06  numero char(15) not null,
07  data_ordem_servico date not null,
08  valor_total float not null,
09  observacao text,
10  constraint pk_osr_idordem_servico primary key (idordem_servico),
11  constraint fk_osr_idcliente foreign key (idcliente) references c_cliente (idcliente),
12  constraint fk_osr_idveiculo foreign key (idveiculo) references c_veiculo
     (idveiculo),
13  constraint fk_osr_idforma_pagamento foreign key (idforma_pagamento)
     references c_forma_pagamento (idforma_pagamento)
14 );
15 create table m_ordem_servico_servico (
16  idordem_servico_servico int not null auto_increment,
17  idordem_servico int not null,
18  idservico int not null,
19  quantidade int not null,
20  valor_unitario decimal(10,2) not null,
21  valor_total float not null,
22  observacao text,
23  constraint pk_oss_idordem_servico_servico primary key
     (idordem_servico_servico),
24  constraint fk_oss_idordem_servico foreign key (idordem_servico)
     references m_ordem_servico (idordem_servico),
25  constraint fk_oss_idservico foreign key (idservico) references c_servico
     (idservico)
26 );
```

Listagem 4. Inserção de dados nas tabelas de movimentação.

```
01 insert into m_ordem_servico (idcliente, idveiculo, idforma_pagamento,
    numero, data_ordem_servico, valor_total)
    values (1, 1, 1, '1234', '2017-02-24', 1000);
02 insert into m_ordem_servico (idcliente, idveiculo, idforma_pagamento,
    numero, data_ordem_servico, valor_total)
    values (2, 2, 1, '4567', '2017-02-25', 1200);
03 insert into m_ordem_servico (idcliente, idveiculo, idforma_pagamento,
    numero, data_ordem_servico, valor_total)
     values (2, 3, 2, '3258', '2017-02-26', 500);
04 insert into m_ordem_servico_servico (idordem_servico, idservico, quantidade,
     valor_unitario, valor_total) values (1, 1, 2, 500, 1000);
05 insert into m_ordem_servico_servico (idordem_servico, idservico, quantidade,
    valor_unitario, valor_total) values (2, 1, 2, 500, 1000);
06 insert into m_ordem_servico_servico (idordem_servico, idservico, quantidade,
    valor_unitario, valor_total) values (2, 2, 1, 200, 200);
07 insert into m_ordem_servico_servico (idordem_servico, idservico, quantidade,
    valor_unitario, valor_total) values (3, 3, 2, 250, 500);
```

Nota

É importante salientar que nos scripts de inserção de dados das **Listagens 2** e **4**, os campos chave primária (idordem_servico e idordem_servico_servico) não foram preenchidos porque estão definidos como autoincremento — o servidor é que será o responsável pela geração dos identificadores.

Criando o Data Module para acesso aos dados

Após a base de dados estar concluída, o próximo passo é criar uma nova aplicação multidispositivo no Delphi (*File > New > Multi-Device Application – Delphi*). Depois, salve o formulário principal com o nome de arquivo *PrincipalFrm* e configure sua propriedade *name* como *frmPrincipal*. Esse tipo de aplicação utiliza o FireMonkey como framework visual e possibilita que as aplicações possam funcionar em Windows, Mac (OSX), iOS e Android.

Agora, crie um Data Module (*File > New > Other > Delphi Files > Data Module*) com o nome de arquivo *ConexaoDtm* e a propriedade *name* com o valor *dtmConexao*. Esse Data Module será utilizado para centralizar o componente de conexão com a base de dados,

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

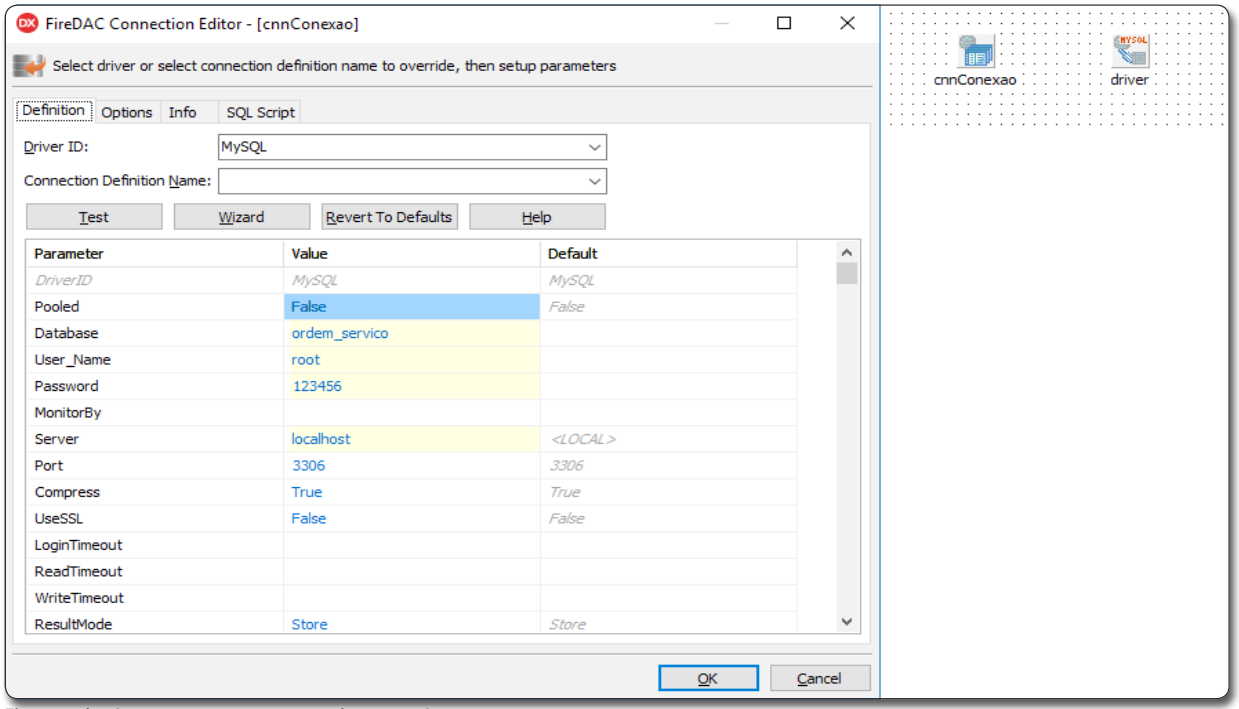

Figura 2. dtmConexao com os componentes driver e cnnConexao

e nele devem ser adicionados dois componentes (vide **Figura 2**): *TFDConnection* (*cnnConexao*) e *TFDPhysMySQLDriverLink* (*driver*), sendo que o primeiro efetivamente realizará a conexão com a base de dados, enquanto que o segundo representa o driver de acesso ao MySQL.

Clique duas vezes sobre o componente de conexão para abrir a janela *FireDAC Connection Editor* (**Figura 3**), na qual as seguintes propriedades devem ser configuras:

- Driver ID: *MySQL*
- Database: *ordem_servico*
- User_Name: *root*
- Password: *123456*
- Server: *localhost*
- Port: *3306*

Essas propriedades equivalem ao nome da base de dados, o usuário do MySQL, a senha (de acordo com o que foi definido na instalação do SGBD), a localização do servidor e a porta na qual os dados irão trafegar. Adicionalmente, a propriedade *LoginPrompt* desse componente deve ser marcada como *False* para que a senha não seja solicitada a cada nova tentativa de conexão.

Com relação ao componente *driver*, a única propriedade que deve ser configurada é a *VendorLib*, e ela deve apontar para o arquivo *libmysql.dll*, que se encontra na pasta de instalação do MySQL.

Criando o menu principal

O formulário principal (*frmPrincipal*) exibe os botões e o menu de acesso para todas as janelas. Para obter uma visualização igual a apresentada na esquerda da **Figura 4**, primeiramente um compo-

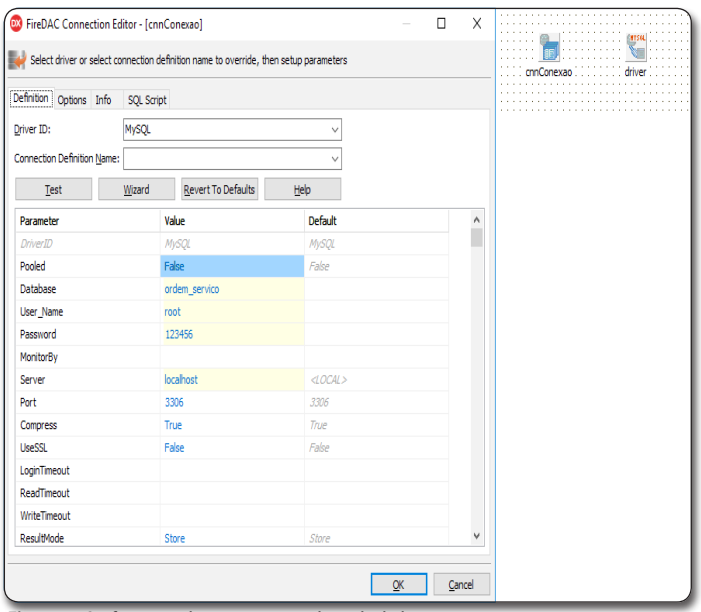

Figura 3. Configuração da conexão com a base de dados

nente *TToolBar* deve ser adicionado (*tlbBarraFerramentas*), dentro do qual devem ser inseridos dois *TButtons* com a propriedade *Align* configurada para *Left* (*btnClientes* e *btnOrdemServico*). Também deve ser adicionado ao formulário um componente *TMainMenu* (*menu*), cujos itens estão apresentados na direita da **Figura 4** (utilize os botões *Add Item* e *Add Child Item* para obter uma lista igual a apresentada na figura). Vale citar que não é possível visualizar em tempo de design o resultado dessa configuração, ou seja, para obter o efeito apresentado o sistema precisa ser executado.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

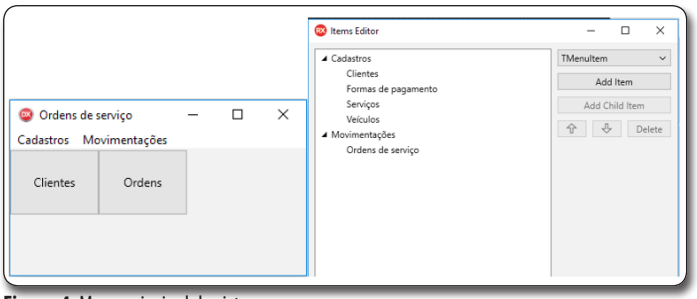

Figura 4. Menu principal do sistema

Quando o usuário clicar em algum botão ou opção de menu, a janela correspondente será aberta na parte central do formulário. Para isso, um componente *TLayout* (*lytPrincipal*) (**BOX 1**) deve ser adicionado no centro da janela com a propriedade *Align* configurada como *Client*, para que, desse modo, ele ocupe toda a janela do formulário principal. Além disso, a propriedade *WindowState* do formulário principal deve ser mudada para *wsMaximized* para a janela ser exibida maximizada quando o sistema for executado.

BOX 1. TLayout

O componente TLayout é um container para outros objetos gráficos e é utilizado quando há a necessidade de organizar diversos componentes gráficos sob o mesmo componente pai, sendo bastante similar às estratégias de layout utilizadas na linguagem Java. Com a maior popularização do desenvolvimento de aplicativos mobile no Delphi, esse componente tornou-se importante para a organização e agrupamento de janelas, e pode ser considerado uma alternativa para o componente TPanel, muito utilizado para a organização das janelas. Consulte a seção Links para mais informações sobre o TLayout.

Nota

Quando se desenvolve aplicações utilizando o FireMonkey, deve-se tomar cuidado com os componentes a serem utilizados, pois alguns podem não estar disponíveis para todos os sistemas operacionais desejados. Uma maneira fácil de identificar a compatibilidade de um componente é posicionar o mouse sobre o nome do mesmo na Tool Palette. Por exemplo, ao posicionar o ponteiro do mouse sobre o componente TMainMenu, um hint surge informando que o mesmo é compatível com 64-bit Windows, 32-bit Windows e OS X. Com isso, podemos chegar à conclusão que esse componente não funcionará em uma aplicação Android ou iOS, por isso é sempre importante, antes de iniciar o desenvolvimento de um projeto, questionar em quais plataformas o software funcionará. Consulte a seção Links para maiores informações sobre menus no FireMonkey.

Um último passo com relação a parte visual da aplicação é a configuração das ações para abertura das janelas, que serão centralizadas no componente *TActionList* (*actAcoes*). Esse componente é bastante útil para manter uma lista de ações, que podem ser utilizadas por botões, menus e outros controles, e tem a vantagem de evitar a repetição de código. Em nosso menu principal, por exemplo, temos um botão e um menu para a abertura da janela de clientes, sendo que o código correspondente para essa ação ficará centralizado nesse componente, tornando desnecessária a replicação de código.

Dê um duplo clique sobre o componente para que o editor de ações seja aberto e adicione as novas ações clicando na opção *New Action* (primeiro botão da janela, sinalizado na **Figura 5**) até que o resultado seja igual ao mostrado na **Figura 5**. Para que as ações fiquem separadas por categorias (*Categories* na figura) é preciso indicar manualmente o nome da categoria na propriedade *Category* de cada ação. A codificação para abertura das janelas será feita posteriormente quando as janelas forem construídas. Por fim, mude a propriedade *Action* tanto das opções de menu quanto dos botões, de modo que eles estejam ligados às suas ações correspondentes.

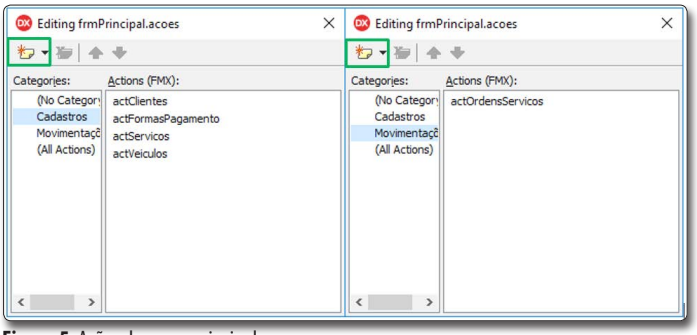

Figura 5. Ações do menu principal

Conforme mencionado anteriormente, as outras janelas do sistema serão abertas dentro do componente TLayout (*lytPrincipal*) quando o usuário executar alguma ação correspondente. Para que isso seja possível, é necessário definir um método que gerenciará os formulários construídos e os exibirá dentro desse layout. A **Listagem 5** apresenta esse código, na qual pode-se notar que foi definida uma variável global do tipo *TForm* (*FFormularioAtivo*) na linha 02 e o procedimento *AbrirJanela* na linha 03, ambos na seção *private* do formulário. Como o próprio nome indica, essa variável é responsável por armazenar o formulário ativo, que o usuário está manipulando no momento.

Na linha 04 é dado início ao método para abrir as janelas. Primeiramente, na linha 08, é verificado se *FFormularioAtivo* possui algum valor (se aponta para algum formulário). Caso sim, na linha 10 é feita uma verificação para saber se o tipo da classe do formulário (*ClassType*) atualmente ativo na memória é igual ao parâmetro *AClasseFormulario*, que possui o formulário que o usuário está tentando abrir. Se essa condição for verdadeira, é um sinal de que o usuário está tentando abrir o mesmo formulário que já está aberto na aplicação. Neste caso, na linha 11 é invocado o método *Exit*, que simplesmente aborta o procedimento para evitar carregar dois objetos iguais na memória.

Caso a condição da linha 10 seja falsa, o fluxo de execução do programa será redirecionado para as linhas 14 e 15, que simplesmente retiram o formulário atualmente ativo da memória. Esse procedimento é feito utilizando o comando *DisposeOf* (linha 14) e atribuindo o valor *nil* para a variável *FFormularioAtivo* (linha 15).

Após as verificações terem sido feitas, na linha 18 da **Listagem 5** o novo formulário é efetivamente criado dentro da aplicação, ou seja, o parâmetro *AClasseFormulário* (que contém o novo formulário a ser aberto) é associado à variável global *FFormularioAtivo* por meio do comando *CreateForm*.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Nota

Na linha 14 da **Listagem 5** foi utilizado o comando DisposeOf para liberar o objeto da memória, mas também é possível utilizar o comando Free. A diferença entre ambos é que o Free passa o controle da liberação do objeto para o Garbage Collector (coletor de lixo) do sistema operacional, enquanto que o DisposeOf já libera a memória no momento em que é chamado. Na maioria dos casos, em aplicações desktop a escolha de qual componente usar não é tão importante, já que nesse tipo de sistema não há tanta preocupação com limitações de hardware. Por outro lado, em aplicações móveis, nas quais existem algumas limitações de hardware, é importante utilizar o DisposeOf para não sobrecarregar o coletor de lixo do sistema operacional do aparelho.

Listagem 5. Método para abertura das janelas.

```
01 private
02  FFormularioAtivo: TForm;
03  procedure AbrirJanela(AClasseFormulario: TComponentClass);
04  procedure TfrmPrincipal.AbrirJanela(AClasseFormulario: TComponentClass);
05 var
06  LayoutBase: TComponent;
07 begin
08  if Assigned(FFormularioAtivo) then
09  begin
10   if FFormularioAtivo.ClassType = AClasseFormulario then
11    Exit
12   else
13   begin
14    FFormularioAtivo.DisposeOf;
15    FFormularioAtivo := nil;
16   end;
17  end;
18  Application.CreateForm(AClasseFormulario, FFormularioAtivo);
19  LayoutBase := FFormularioAtivo.FindComponent('lytBase');
20  if Assigned(LayoutBase) then
21   lytPrincipal.AddObject(TLayout(LayoutBase));
22 end;
```

Na linha 19 é utilizada a variável *LayoutBase* do tipo *TComponent* (declarada na linha 06), sendo que ela recebe um layout que estará presente no novo formulário que será criado. Conforme explanado anteriormente, as novas janelas serão abertas dentro do componente *lytPrincipal* do formulário principal, porém o que efetivamente deve ser adicionado dentro de um layout é um outro layout, e não um formulário. Para entender melhor essa questão, a parte superior da **Figura 6** apresenta a estrutura da janela principal, na qual existe um componente *TLayout* (*lytPrincipal*) dentro do formulário (*TForm*). Similarmente, na parte inferior dessa figura é possível observar que há um formulário de cadastro (*TForm*) e dentro dele existe outro *TLayout* (*lytBase*), dentro do qual ficarão armazenados todos os componentes para manipulação dos registros, bem como botões, edits e grids. Note também que a flecha indica que tudo o que estiver dentro do *lytBase* será copiado para o *lytPrincipal*, ou seja, se algum componente estiver fora do layout não será carregado.

A **Figura 6** explicou a função da linha 19 da **Listagem 5**, que é: o método *FindComponent* fará uma pesquisa no formulário ativo e localizará um componente com o nome *lytBase*, copiando-o para a variável *LayoutBase*. Como o nome *lytBase* é constante neste exemplo, todos os demais formulários deverão possuir um layout exatamente com esse mesmo nome, caso contrário os componentes não serão copiados. Por fim, na linha 20 é verificado se *LayoutBase* possui algum valor e, na linha 21, o layout do novo formulário é adicionado ao layout do formulário principal, código que representa a flecha na **Figura 6**.

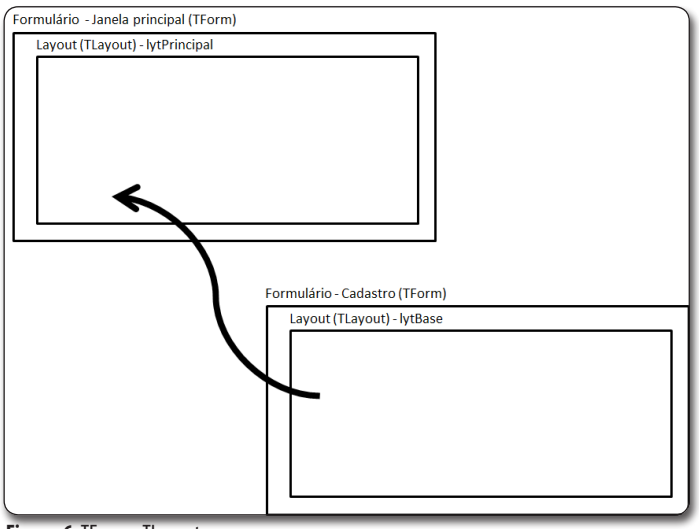

Figura 6. TForm e TLayout

Nota

Neste artigo optou-se pelo desenvolvimento da janela principal utilizando a abertura de uma tela por vez, que tem sido o padrão utilizado em grande parte das aplicações mais modernas. Consulte a seção **Links** para mais informações sobre aplicações MDI (Multiple Document Interface) no Delphi utilizando a VCL (Visual Component Library).

Criando o formulário padrão para listagem de registros

Quando o usuário clicar em alguma das opções do menu, primeiramente será carregado no formulário principal uma listagem completa de todos os registros correspondentes à respectiva tabela. Para evitar repetição de componentes no projeto, criaremos uma janela padrão (vide **Figura 7**) com conceitos de herança que servirá de base para todas as outras, já que as funcionalidades serão as mesmas tanto para as tabelas de cadastro quanto para as de movimentação.

Seguindo a opção de menu *File* > *New* > *Multi-Device Form – Delphi* será criado um novo formulário FireMonkey, o qual deve ser salvo com o nome *ListagemPadraoFrm* e ter sua propriedade *name* mudada para *frmListagemPadrao*. Para obter uma visualização semelhante à **Figura 7**, inicialmente deve ser adicionado um componente *TLayout* com o nome *lytBase*, o qual necessita ser exibido na janeira inteira (*Align* = *Client*) e servir como contêiner para os demais. Depois, uma *TToolBar* (*tblBarraFerramentas*) necessita ser adicionada ao topo com três *TButtons* (*btnIncluir*, *btnAlterar* e *btnExcluir*) com a propriedade *Align* configurada com o valor *Left*. Esses botões serão utilizados para abrir uma nova janela para inclusão e alteração e também para excluir o registro que estiver selecionado.

Abaixo da barra de ferramentas encontra-se um componente *TPanel* (*pnlPesquisa*) com a propriedade *Align* configurada para

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

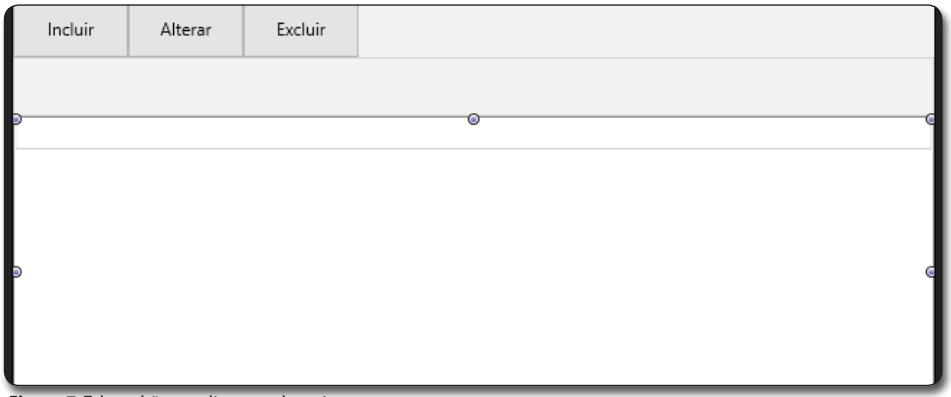

Figura 7. Tela padrão para listagem de registros

Top, o qual servirá posteriormente para a adição de componentes específicos para a realização de consultas SQL personalizadas. Por fim, encontra-se um componente *TGrid* (*grdDados*), que apresentará os registros retornados da base de dados. A vantagem de utilizar uma janela padrão é que os componentes visuais e o código fonte podem ser reaproveitados em outras telas, o que facilita a manutenção do software bem como a adição de novas funcionalidades, que podem ser automaticamente replicadas para todas as outras janelas.

Nota

No FireMonkey não existe o componente TDBGrid, que é um dos mais utilizados para listagem de dados em aplicações desktop utilizando a VCL (Visual Component Library). Como alternativa, pode-se utilizar o TGrid ou o TListView.

Criando a listagem de ordens de serviço

Agora que temos nosso formulário padrão para a listagem de registros, construiremos a listagem de todas as ordens de serviço cadastradas na base de dados. A estrutura do projeto para cada janela será composta por dois arquivos: um novo Data Module, que conterá a consulta SQL para o retorno dos registros, e uma nova janela, que herdará da listagem padrão.

A **Figura 8** apresenta o novo Data Module (*File* > *New* > *Other* > *Delphi Files* > *Data Module*), que deve ser salvo com o nome *OrdemServicoDtm*, ter sua propriedade *name* alterada para *dtmOrdemServico* e importar o arquivo *ConexaoDtm* para que tenha acesso ao componente de conexão (*File* > *Use Unit*). Note que o único componente adicionado é um *TFDQuery* (*qryOrdemServico*), que será responsável por executar a instrução SQL. As seguintes propriedades devem ser configuradas para esse componente:

- Connection: dtmConexao.cnnConexao
- SQL: select ors.idordem_servico, cln.nome as cliente, vlc.placa as veiculo, fpg.nome as forma_pagamento, ors.numero, ors.data_ordem_servico from m_ordem_servico ors inner join c_cliente cln on ors.idcliente = cln.idcliente inner join c_veiculo vlc on ors.idveiculo = vlc.idveiculo inner join c_forma_pagamento fpg on ors.idforma_pagamento = fpg.idforma_pagamento

A primeira propriedade liga o componente à conexão com a base de dados, enquanto que a segunda apresenta um código SQL completo para buscar todas as ordens de serviço, juntamente com o nome do cliente, a placa do veículo e a forma de pagamento, informações essas que estão armazenados em suas respectivas tabelas. A seguir deve-se acessar o *Fields Editor* da query (botão direito no componente) e escolher a opção *Add All Fields* para que todos os campos sejam adicionados, conforme mostra a **Figura 8**.

Neste exemplo, somente os campos *observação* e *valor_total* não estão sendo retornados, porém, em uma aplicação comercial que recebe muitos acessos simultâneos, uma boa prática é adicionar na instrução SQL somente os campos principais e que efetivamente serão úteis para consulta, evitando, assim, possíveis problemas de tráfego na rede. Outra questão é que nesta consulta não estão sendo retornados os registros detalhe que pertencem à tabela *m_ordem_servico_servico*, pois isso também poderia acarretar em atrasos na rede para a busca dos dados. Em suma, os serviços da ordem de serviço serão carregados somente quando o usuário clicar no botão *btnAlterar*, sendo aberta uma nova janela para edição dos registros.

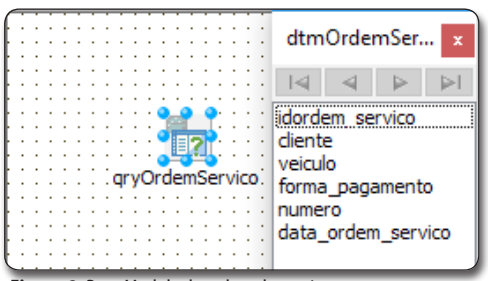

Figura 8. Data Module da ordem de serviço

O próximo passo é criar o formulário para listagem dos registros, o qual será baseado na listagem padrão construída anteriormente. Para isso, deve-se seguir a opção de menu *File* > *New* > *Other* > *Inheritable Items* e escolher o item *frmListagemPadrao*, para que o conceito de herança seja utilizado e nosso novo formulário herde todas as características de seu ancestral. A **Figura 9** mostra essa janela, na qual todos os formulários e/ou Data Modules já criados no projeto podem ser herdados.

A nova janela deve ser salva com o nome *OrdemServicoListagemFrm*, e seu nome (propriedade *name*) deve ser alterado para *frmOrdemServicoListagem*. Note que a janela possui o mesmo visual da listagem padrão, e para que os registros sejam exibidos não será necessária nenhuma linha de código, basta que esse novo formulário tenha acesso ao *dtmOrdemServico* (*File* > *Use Unit*).

A vinculação do grid com os dados será feita por meio do recurso de LiveBindings (*View* > *LiveBindings Designer*). Clicando no botão do wizard dessa janela, deve-se escolher a opção *Link*

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

a grid with a data source na primeira etapa do assistente, selecionar a grade na etapa 2 (esquerda da **Figura 10**) e selecionar a query na terceira etapa (direita da **Figura 10**). Na quarta e última etapa do assistente é possível adicionar um componente para navegação - *Add data source navigator* -, mas para esta aplicação essa opção não precisa ser adicionada. Após a conclusão do assistente são adicionados dois componentes: *BindingsList1* e *BindSourceDB1*, que são os responsáveis pela operacionalização da vinculação entre os componentes. A partir deste momento, quando a query for ativada (propriedade *Active = True*) os dados já podem ser visualizados na grade.

Para que a tela seja exibida em tempo de execução pelo menu principal, é necessário, primeiramente, que a janela principal (*PrincipalFrm*) tenha acesso à listagem das ordens de serviço, o que pode ser configurado pelo comando *File > Use Unit*. Depois, basta que o evento *OnExecute* da ação *actOrdensServicos* seja codificado de acordo com o que é mostrado na **Listagem 6**. O único comando é mostrado na linha 03, na qual é invocado o método para abertura das janelas definido anteriormente, passando como parâmetro a janela de listagem das ordens de serviço.

Listagem 6. Abertura da listagem das ordens de serviços.

```
01 procedure TfrmPrincipal.actOrdensServicos
     Execute(Sender: TObject);
02 begin
03   AbrirJanela(TfrmOrdemServicoListagem);
04 end;
```

A **Figura 11** mostra um exemplo de execução, no qual se pode observar as três ordens de serviço que foram anteriormente adicionadas via SQL.

É importante lembrar que o componente *qryOrdemServico* necessita estar ativo para que os dados sejam exibidos em tela, porém essa não é uma boa prática para ser mantida em sistemas comerciais.

O recomendável é preservar as queries e o componente de conexão sempre fechados e abri-los somente quando uma requisição for necessária em tempo de execução.

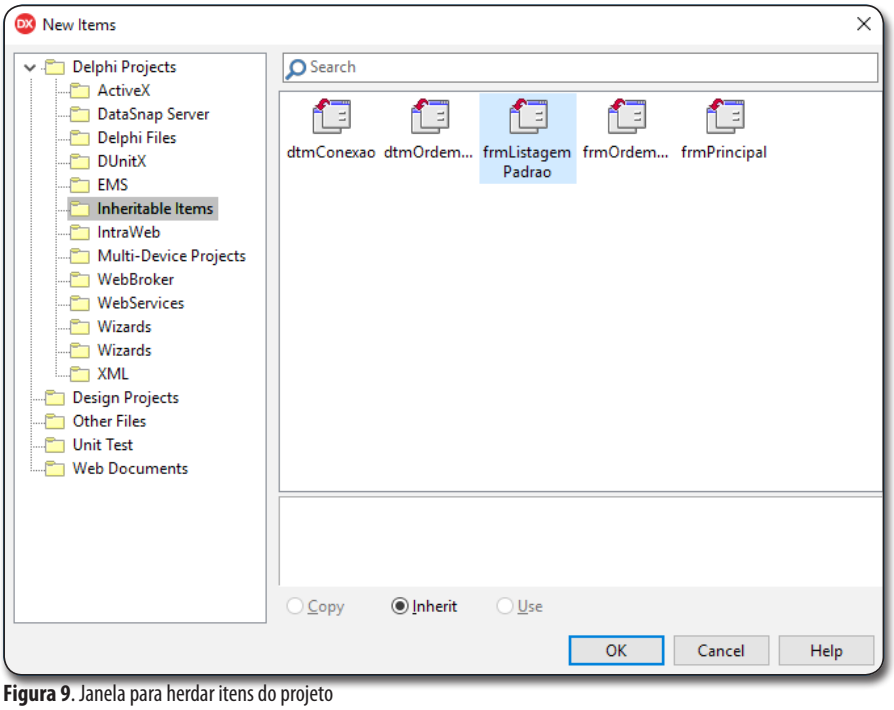

Figura 9. Janela para herdar itens do projeto

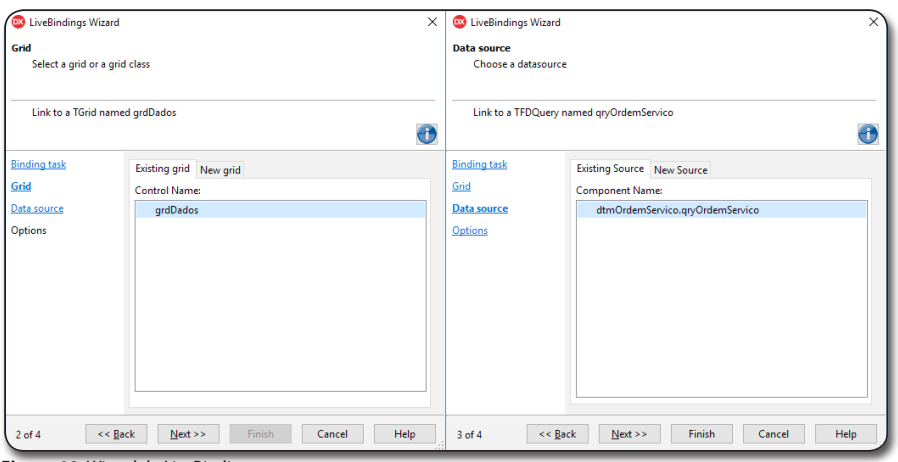

Figura 10. Wizard do LiveBindings

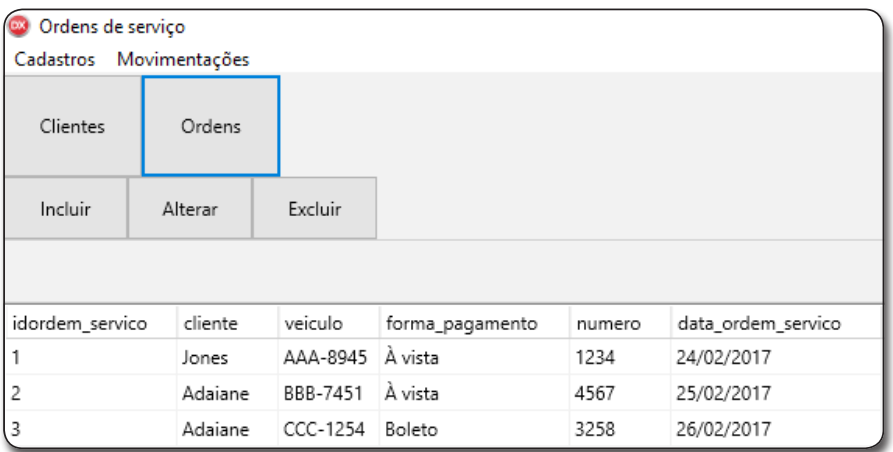

Figura 11. Execução da listagem de ordens de serviço

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Além disso, não é também uma boa prática carregar todos os registros quando as janelas são abertas, pois em uma tabela de movimentação podem existir milhares de linhas e elas poderão demorar um tempo considerável para serem carregadas, fora o fato de que dificilmente o usuário precisará visualizar todos os registros de uma única vez.

Para resolver esse problema adicionaremos um *TEdit* (*edtNumero*) e um *TButton* (*btnPesquisar*) no painel localizado abaixo dos botões de inclusão, alteração e exclusão (**Figura 12**). O objetivo é que o usuário informe o código da ordem de serviço e a consulta SQL seja executada com um filtro *where* que busque somente o código informado.

A **Listagem 7** apresenta a codificação, que deve ser inserida no evento *OnClick* do novo botão. Na linha 03 é definida uma variável do tipo string (*ASql*), que receberá, entre as linhas 06 e 10, o mesmo comando SQL que foi definido no componente *qryOrdemServico*. Na linha 11 é feita uma estrutura condicional que verifica se existe algum valor no edit. Caso afirmativo, na linha 12 é concatenado um comando *where* para fazer o filtro. Por fim, na linha 13 a query recebe a consulta completa e na linha 14 ela é aberta, momento no qual ela será executada. A **Figura 12** apresenta um exemplo de execução com uma consulta sendo feita pelo código "1234".

Criando as telas de listagem dos cadastros

Os mesmos processos devem ser executados para a criação das outras listagens, ou seja: definição de um Data Module para cada cadastro, criação de janelas utilizando herança, vinculação via LiveBindings e programação das ações na janela principal. Serão mostrados na sequência somente alguns desses passos, visto que os procedimentos são exatamente os mesmos, com a única exceção de que para esses cadastros básicos não utilizaremos o recurso de pesquisa mostrado na **Listagem 7**. A **Figura 13** mostra os quatro Data Modules (um para cada tabela) juntamente com os respectivos *Fields Editors*.

Listagem 7. Abertura da listagem das ordens de serviços.

```
01 procedure TfrmOrdemServicoListagem.btnPesquisarClick(Sender: TObject);
02 var
03   ASql: String;
04 begin
05   inherited;
06   ASql := 'select ors.idordem_servico, cln.nome as cliente, vlc.placa as veiculo,
       fpg.nome as forma_pagamento,';
07   ASql := ASql + ' ors.numero, ors.data_ordem_servico from
       m_ordem_servico ors ';
08   ASql := ASql + ' inner join c_cliente cln on ors.idcliente = cln.idcliente ';
09   ASql := ASql + ' inner join c_veiculo vlc on ors.idveiculo = vlc.idveiculo ';
10   ASql := ASql + ' inner join c_forma_pagamento fpg on ors.idforma
       _pagamento = fpg.idforma_pagamento';
11   if Trim(edtNumero.Text) <> '' then
12     ASql := ASql + ' where ors.numero = ' + QuotedStr(edtNumero.Text);
13   dtmOrdemServico.qryOrdemServico.SQL.Text := ASql;
14   dtmOrdemServico.qryOrdemServico.Open();
15 end;
```

Nota

O comando Trim na linha 11 da **Listagem 7** é utilizado para retirar os espaços em branco do início e do final da string, enquanto que a função QuotedStr da linha 12 adiciona aspas simples no início e no final do texto para não serem gerados erros de SQL.

A seguir serão mostrados os comandos SQL que devem ser inseridos na propriedade *SQL* de cada um dos componentes TFDQuery.

- *qryCliente*: select idcliente, nome, data_nascimento, cpf, rg from c_pessoa
- *qryFormaPagamento*: select idforma_pagamento, nome from c_forma_pagamento
- *qryServico*: select idservico, nome from c_servico
- *qryVeiculo*: select idveiculo, placa, modelo from c_veiculo

Para finalizar a criação das janelas, o próximo passo é a criação de quatro formulários (*ClienteListagemFrm*, *FormaPagamentoListagemFrm*, *ServicoListagemFrm* e *VeiculoListagemFrm*) herdando de *ListagemPadraoFrm*. Após isso, basta seguir o wizard para configuração dos LiveBindings para que os registros possam ser visualizados no componente *TGrid* (*grdDados*). Por fim, a **Listagem 8** mostra o código que deve ser implementado no evento *OnExecute* de cada

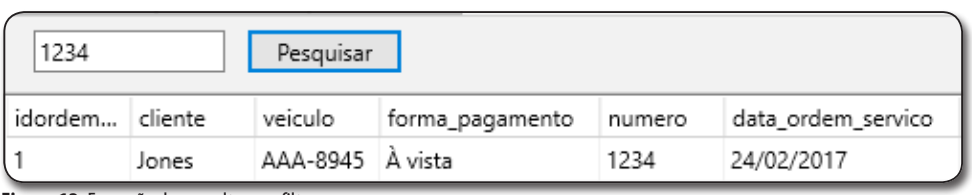

Figura 12. Execução da consulta com filtros

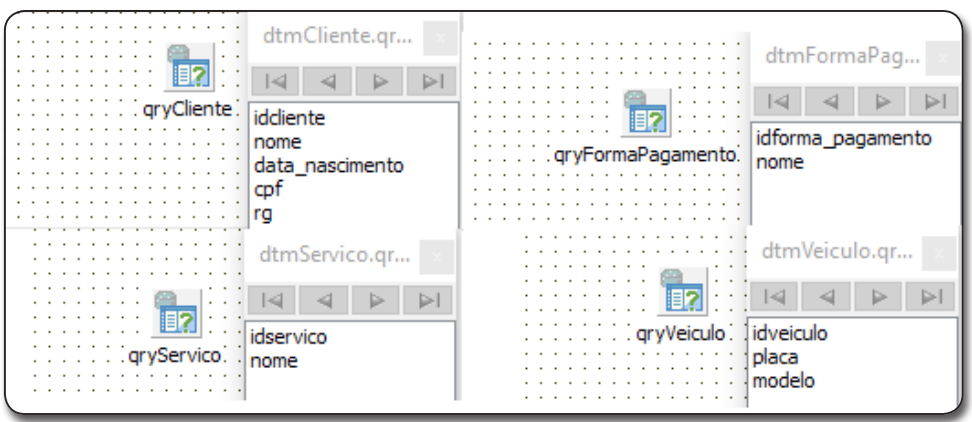

Figura 13. Data Modules para os cadastros

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

uma das ações, o qual simplesmente invoca o método *AbrirJanela* e passa como parâmetro a tela que deve ser aberta.

Com isso, a primeira parte do software já está totalmente funcional no que diz respeito aos recursos implementados. Uma dúvida comum é se realmente são necessários todos esses Data Modules somente para manter as queries com os SQLs de consulta aos dados.

Listagem 8. Codificação das ações da janela principal.

```
01 procedure TfrmPrincipal.actClientesExecute(Sender: TObject);
02 begin
03   AbrirJanela(TfrmClienteListagem);
04 end;
05 procedure TfrmPrincipal.actFormasPagamentoExecute(Sender: TObject);
06 begin
07   AbrirJanela(TfrmFormaPagamentoListagem);
08 end;
09 procedure TfrmPrincipal.actServicosExecute(Sender: TObject);
10 begin
11   AbrirJanela(TfrmServicoListagem);
12 end;
13 procedure TfrmPrincipal.actVeiculosExecute(Sender: TObject);
14 begin
15   AbrirJanela(TfrmVeiculoListagem);
16 end;
```

Nota

É preciso enfatizar que até este ponto do software, com exceção da janela de ordens de serviço, todas as outras necessitam que a query esteja aberta em design time para que os dados sejam visualizados, visto que nenhum código adicional foi implementado. Para resolver essa questão o seguinte código pode ser adicionado ao evento OnCreate do formulário de clientes, por exemplo: dtmCliente.qryCliente.Open(). O mesmo código pode ser implementado para os outros cadastros, bastando mudar o nome do Data Module e da query.

Apesar de serem vários arquivos, uma boa prática é que a estrutura do projeto fique consistente e com os arquivos divididos exatamente com suas funções específicas. Por exemplo, caso fosse definido somente um Data Module para centralizar todos os componentes, um futuro crescimento do sistema ficaria mais difícil de ser implementado, visto que os componentes ficariam "acoplados" em um único arquivo. Neste contexto, haveria a necessidade de dividi-los em arquivos separados assim como o feito neste artigo. Outra desvantagem de uni-los em um único local é que a codificação ficaria misturada entre vários contextos diferentes, e dependendo do tamanho do software seria difícil realizar manutenções. Por fim, a divisão do software em módulos distintos que poderiam ser acoplados em outros softwares também ficaria prejudicada, pois, como já mencionado, as funções de vários contextos ficariam misturadas.

Autor

Jones Granatyr

Doutorando em Informática bolsista CAPES e Mestre em Ciência da Computação bolsista CNPq, ambos na área de Inteligência Artificial. É fundador do portal IA Expert e trabalha em projetos de pesquisa relacionados à área de Inteligência Artificial, tais como Sistemas Especialistas, Mineração de Dados, Mineração de Textos e Sistemas Multiagente.

Links:

Auto-Incremental Fields (FireDAC)
http://docwiki.embarcadero.com/RADStudio/Berlin/en/Auto-Incremental_Fields_(FireDAC)

Caching Updates (FireDAC)
http://docwiki.embarcadero.com/RADStudio/Berlin/en/Caching_Updates_(FireDAC)

FireMonkey Layouts Strategies
http://docwiki.embarcadero.com/RADStudio/Seattle/en/FireMonkey_Layouts_Strategies

MDI Applications
http://docwiki.embarcadero.com/RADStudio/Seattle/en/MDI_Applications

TActionList
http://docwiki.embarcadero.com/Libraries/XE7/en/FMX.ActnList.TActionList

TGrid
http://docwiki.embarcadero.com/Libraries/Berlin/en/FMX.Grid.TGrid

Using Menus in a FireMonkey Application
http://docwiki.embarcadero.com/RADStudio/Seattle/en/Using_Menus_in_a_FireMonkey_Application

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,
assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486